版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

Broadview®
www.broadview.com.cn

前端撷英馆

React与Redux开发实例精解



刘一奇 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介



刘一奇，曾在ESRI中国、中科院计算所工作学习，现供职于奇虎360，担任前端工程师。工作之余，他热衷于开源，是Redux拓展redux-amrc的作者，同时也是一位盛产的博主，写过React、Redux、Angular2的博客教程，广受读者欢迎。

前端撷英馆

React与Redux开发实例精解



刘一奇 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书共分四部分，全面介绍如何使用 React 与 Redux 进行 Web 开发。第一部分是基础篇，介绍 React 与 Redux 的基础知识；第二部分是进阶篇，通过精彩的官方示例学习 React 与 Redux；第三部分是拓展篇，主要学习一些优秀的第三方拓展；第四部分是实战篇，将会带领读者一步步搭建大型 Web 应用程序。

本书适合熟悉 JavaScript 编程，有意使用 React 与 Redux 搭建 Web 应用的程序员学习参考。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

React 与 Redux 开发实例精解 / 刘一奇著.—北京：电子工业出版社，2016.11
（前端撷英馆）
ISBN 978-7-121-30267-1
I. ① R…II. ① 刘…III. ① JAVA 语言 – 程序设计 IV. ① TP312.8

中国版本图书馆 CIP 数据核字（2016）第 265104 号

责任编辑：许　艳
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：20　　字数：410 千字
版　　次：2016 年 11 月第 1 版
印　　次：2016 年 11 月第 1 次印刷
定　　价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819　faq@phei.com.cn。

致读者

这本书是为想要用 React、Redux、Babel 和 Webpack 搭建 Web 应用程序的程序员准备的。与其他技术栈相比，这个技术栈（指的是 React、Redux、Babel、Webpack 以及社区中众多与此相关的优秀工具）的学习成本并不低，但是其中的智慧与思想却着实令人赞叹：清爽的组件代码结束了满屏 HTML 标签与类名的噩梦；声明式地将数据映射为界面减少了烦琐的 DOM 操作；高阶函数、管道、柯里化等函数式编程思想的运用使得复杂逻辑的处理变得相当简单；自动化构建编译技术降低了源代码与静态资源的管理成本。随着时间的推移，技术工具或许很快会更新换代，但是这些前端工程中的最佳实践思想却不会轻易过时。如果你不仅想学习这些技术的使用方法，更想了解这些技术背后所包含的思想与智慧，那么我真诚地建议你阅读本书。

阅读本书之前的准备

想要很好地理解本书的内容，你需要具备一定的 Node.js 和 ES2015、ES2016 基础，至少要了解一些常见的概念，例如 NPM、模块系统、常量、函数、装饰器和类。除此之外，你最好还熟悉 JavaScript 程序在服务器和浏览器环境下的调试技术。

学习本书示例代码时，大部分时间都要在命令行中进行操作。因此，你应该能够熟练使用自己操作系统中的命令行工具。

本书附带的程序是开源的，全部上传到了 GitHub。这些示例程序中的开发经验来自开源社区和笔者自己的实践思考。我不认为这是使用 React 与 Redux 开发 Web 应用程序的唯一方式。你可以将其作为参考，如果有更好的想法，强烈建议你将其贡献给社区，与更多的开发者一起交流进步。

最后要说明的是，本书因篇幅有限等原因，无法对每一项技术的讲解都做到完整而详尽。因此，你还需要根据书中的提示和推荐，去阅读参考相应技术的官方文档。

本书的内容安排

本书分为四部分。

第一部分　基础篇　主要介绍 React 与 Redux 的基础知识。

- 第 1 章　简要介绍本书所要讲解的技术，包括 React、Redux、Node 与 Universal 渲染、Babel 及 Webpack。
- 第 2 章　讲解如何在 Node.js 中运行 React。
- 第 3 章　讲解如何在浏览器中运行 React。
- 第 4 章　介绍开发服务器和热替换技术及其实现过程。
- 第 5 章　介绍 React 的创新语法：JSX。
- 第 6 章　介绍 React 的数据载体：state、props 与 context。
- 第 7 章　介绍 React 的两个对象：ReactElement 和组件实例。
- 第 8 章　介绍 Redux 的 action、reducer 与 store。
- 第 9 章　介绍 Redux 的 action 创建函数和 Thunk 中间件。

第二部分　进阶篇　通过精彩的官方示例学习 React 与 Redux。

- 第 10 章　讲解如何手动将 Redux 连接到 React 中。
- 第 11 章　讲解如何使用 react-redux 将 Redux 连接到 React 中。
- 第 12 章　介绍 Redux 的撤销/重做和 Redux 开发者工具的使用。
- 第 13 章　讲解如何在 React 与 Redux 的程序中编写测试。
- 第 14 章　讨论 Redux 的全局状态与 React 组件的内部状态。
- 第 15 章　讲解 React 与 Redux 中常用的几种数组处理方法。
- 第 16 章　介绍 Redux 中的异步处理。
- 第 17 章　介绍如何自定义 Redux 中间件。
- 第 18 章　讲解如何使用 React 与 Redux 实现 Universal 渲染。

第三部分　拓展篇　学习一些优秀的第三方拓展。

- 第 19 章　介绍 Webpack 同构工具。
- 第 20 章　介绍 React Router，并通过它实现 Universal 路由。
- 第 21 章　介绍多页面下的异步处理。
- 第 22 章　介绍了 bootstrap-loader、PostCSS、Autoprefixer 和 React-Bootstrap。

第四部分　实战篇　一步步搭建大型 Web 应用程序。

- 第 23 章　介绍一个大型项目的基本结构。
- 第 24 章　使用 ReduxForm 制作各种表单。
- 第 25 章　实现图表与表格。
- 第 26 章　实现用户认证。
- 第 27 章　讲解部署到 Heroku 的方法，并实现持续集成。
- 第 28 章　列出其他资源。

如何使用示例代码

本书使用的示例代码可在 GitHub 上获取：https://github.com/lewis617/react-redux-book 和 https://github.com/lewis617/react-redux-heroku。

除了第 1 章和第 28 章外，本书每个章节都有一个示例程序。前三部分使用了短而精的示例代码，独立地演示技术的功能。从第四部分开始，每一章的例子都会建立在前一章例子的基础上，最后完成一个大型项目的搭建。

另外，为了方便部署到 Heroku，并实现持续集成，第 27 章的例子被独立到了另一个 GitHub 仓库中，也就是上述的第二个 GitHub 仓库。

致谢

在写这本书的过程中，家人、朋友、领导、同事都给我了很多帮助。没有他们，我是不可能完成本书的，在此表示由衷的感谢。

首先我要特别感谢王祖超、梁锦津、宋兵、陈仕杰、屈光宇、吕明岩，他们在本书撰写的不同阶段审阅了书稿，并对书稿涵盖的内容和组织方式给予了建设性建议。

还要感谢电子工业出版社的许艳编辑，她对本书进行了出色平稳的编辑，并给予我无尽的支持、建议和协助。

最后，感谢在 GitHub 社区中为本书提供帮助的人：

- Dan Abramov（Redux 的作者）
- Nikolay Nikolaev（Webpack 同构工具的作者）
- Erik Rasmussen（ReduxForm 的作者）
- Eric Ferraiuolo（serialize-javascript 的贡献者）
- Mihail Diordiev（redux-devtools-extension 的作者）

他们积极热心地回复我提交的 Issue，为本书提供了重要的技术支持。

目录

第一部分　基础篇　1

第 1 章　技术简介 2

React　2

Redux　3

Node 与 Universal 渲染　4

Babel　5

Webpack　5

总结　6

第 2 章　在 Node.js 中运行 React 7

例子　7

编写 React 组件　7

在 Node.js 中渲染组件　8

使用 Babel 编译运行 Node.js 程序　9

Require Hook 简介　9

使用 Require Hook 的步骤　9

总结　10

第 3 章 在浏览器中运行 React 11

例子 11

组件复用 11

在浏览器中渲染 React 组件 12

使用 Webpack 打包编译 12

在浏览器中运行 14

总结 14

第 4 章 开发服务器和热替换 15

例子 15

安装 16

配置 Babel 16

使用 react-hmre 预设 16

react-hmre 的功能 17

配置 Webpack 18

配置 Express 服务器 21

总结 22

第 5 章 React 的创新语法：JSX 24

例子 24

JSX 简介 24

JSX 常用语法 25

类似 HTML 25

JavaScript 表达式 26

样式 27

注释 27

数组 27

HTML 标签 vs. React 组件 28

总结 29

第 6 章 React 的数据载体：state、props 与 context 30

例子 30

State 31

Props 32

使用 props 33

验证 props 33

组合使用 state 与 props 35

Context 36

使用 props 传递数据 36

使用 context 传递数据 38

Props 与 context 的适用场景 40

在 React 开发者工具中查看 41

总结 42

第 7 章 React 的两个对象：ReactElement 与组件实例 43

例子 43

ReactElement 44

JSX 中的闭合标签是 ReactElement 44

ReactElement 是什么 44

ReactElement 的两种类型 45

React 组件的渲染流程 45

组件实例 46

组件实例简介 46

组件、ReactElement 与组件实例的区别 47

组件实例的生灭：生命周期函数 48

React 组件中的 this 49

总结 55

第 8 章 初识 Redux 56

例子 56

Action 56
Reducer 57
定义 57
纯函数 58
不能修改参数 state 58
Store 60
职能 60
创建 60
获取与监听 60
发起 action 61
总结 62

第 9 章 Action 创建函数与 Redux Thunk 中间件 64
例子 64
Action 创建函数 65
编写 65
发起 65
意义 66
Redux Thunk 中间件 66
功能 66
安装激活 67
总结 69

第二部分 进阶篇 71

第 10 章 React 与 Redux 的连接：手动连接 72
例子 72
手动连接的步骤 73
编写 React 组件 73
编写 Redux 75

将 Redux 手动连接到 React 组件 76
手动连接的缺点 77
总结 77

第 11 章 React 与 Redux 的连接：使用 react-redux 连接 78
例子 78
连接步骤 78
多种写法 81
连接原理 85
Provider 工作原理 85
connect 工作原理 86
展示组件与容器组件 86
总结 87

第 12 章 实现撤销/重做 88
例子 88
撤销/重做 88
Redux 开发者工具 92
安装 92
使用 93
Redux 并不“低效” 97
总结 98

第 13 章 测试 99
例子 99
测试工具 100
Mocha 100
Enzyme 101
Expect 102
测试 action 创建函数 103

测试 React 组件 106
测试容器组件 108
测试 reducer 纯函数 110
运行测试脚本 111
总结 112

第 14 章 Redux 的全局状态与 React 组件的内部状态 114

例子 114
全局状态与内部状态的定义 114
全局状态完全替代了内部状态? 115
全局状态与内部状态的适用场景 118
总结 118

第 15 章 React 与 Redux 中的数组处理 119

例子 119
reduce() 119
概述 119
语法 120
示例 120
filter() 121
概述 121
语法 121
示例 122
map() 122
概述 122
语法 122
示例 123
every() 123
概述 123
语法 123

示例 124
some() 124
概述 124
语法 124
示例 125
展开运算符 125
概述 125
语法 125
示例 125
总结 126

第 16 章 Redux 的大舞台：异步 127
例子 127
JavaScript 事件驱动编程 127
Promise 129
Redux 中的异步 132
异步 action 创建函数 133
异步请求前后的 state 变化 133
State tree 结构 135
查看 state tree 136
如何构建 state tree 136
异步与变化 139
准备工作 139
追踪异步与变化 143
总结 145

第 17 章 自定义 Redux 中间件 147
例子 147
中间件的功能 147
中间件的编写 148

中间件的执行 149
自定义 API 中间件 151
总结 154

第 18 章 Universal 渲染 155
例子 155
公用一套代码 156
服务端渲染 156
客户端渲染 158
总结 159

第三部分 拓展篇 161

第 19 章 Universal 渲染神器：Webpack 同构工具 162
例子 162
Webpack 同构工具的功能 163
服务端调用 164
客户端调用 165
真实场景 167
配置服务器的入口 167
独立开发服务器 168
配置 Webpack 169
配置 Webpack 同构工具 171
在服务端和客户端使用 require() 173
总结 174

第 20 章 多页面的实现：路由 175
例子 175
路由也是组件 176
路由匹配 177

使用 Link 和 IndexLink 导航 178
服务端路由 179
总结 181

第 21 章 多页面下的异步操作 182
例子 182
redux-amrc 182
功能 183
配置 184
使用 184
Action 与 state 185
API 186
多页面下的异步操作 187
编写测试 API 187
预载数据 188
手动加载数据 191
操作数据 192
总结 193

第 22 章 使用 Bootstrap 194
例子 194
bootstrap-loader 195
基本用法 195
配置 196
PostCSS 与 Autoprefixer 200
简介 200
用法 201
React-Bootstrap 202
总结 204

第四部分 实战篇 205

第 23 章 搭建大型项目 206

例子 206

开发服务器 209

开发环境下的 Webpack 配置 209

Webpack 同构工具的配置 212

独立的开发服务器 213

启动开发服务器 214

前端服务器 215

配置前端服务器 215

使用组件渲染 HTML 页面 218

启动前端服务器 221

API 服务器 222

配置 API 服务器 222

启动 API 服务器 223

生产环境下的构建编译 224

编译运行 Node.js 224

生产环境下的 Webpack 配置 225

公用代码 229

工具集 229

路由与页面 231

质量保证 235

测试 235

Airbnb 编码规范 239

去毛机 246

EditorConfig 248

总结 248

第 24 章　表单 250

例子 250

高阶组件 ReduxForm 251

设置 252

简单表单 253

同步验证表单 256

异步失焦验证表单 258

提交表单到服务器 260

总结 264

第 25 章　图表与表格 265

例子 265

图表组件 265

表格组件 269

从服务器获取数据 272

总结 275

第 26 章　用户认证 277

例子 277

编写认证 API 279

Express-session 中间件 279

加载当前认证状态 280

登录与登出 280

API 权限保护 281

发送 cookie 282

编写用于认证的 action 创建函数 283

登录页面和导航栏 285

前端路由保护 290

总结 291

第 27 章 部署 292

例子 292

Heroku 平台 293

安装工具并登录 293

准备程序 293

部署程序 294

其他操作 295

持续集成 296

与生产相关的问题 298

总结 299

第 28 章 其他资源 300

在线文档 300

参与社区活动 301

第一部分

基础篇

第1章

技术简介

React

毫不夸张地说，近年来最让全球 JavaScript 开发者感到疯狂的事情就是 React 的横空出世。作为一个构建用户界面的工具库，React 被众多知名公司所使用，包括 Facebook、Airbnb、Uber、Alibaba 等。如果你想进入这些公司工作，那么你应该学习 React。

React 官方是这样介绍它的："一个声明式、高效、灵活的、创建用户界面的 JavaScript 库。"这究竟是什么意思呢？下面我们来逐一解读。

声明式是指只要使用 React 描述组件的样子就可以改变用户界面。传统方法是命令式地操作 DOM，不仅需要记住大量的 API，而且还会增加代码的耦合度，使得项目难以维护。React 使用自己的魔法改变界面，让开发者最大限度地少与 DOM 交互，降低了处理视图的难度。

高效主要得益于 React 的虚拟 DOM，以及其 Diff 算法。传统的 Diff 算法的复杂度是极高的，而 Facebook 的工程师巧妙地结合 Web 界面的特点改进了 Diff 算法，使其性能出众。拥有了高性能的 Diff 算法，我们就可以在"刷新"全部界面时只更新需要改变的那部分界面了。也正因为这个原因，React 可以使用单向数据流改变界面，抛弃了传统的数据绑定，减少了样板代码和重量。

灵活是指 React 可以作为视图层与其他技术栈配合使用，比如代替 Angular 的指令，或者与 Redux 搭配，等等。

除了上面介绍的这些优点外，React 最让人兴奋的莫过于 Universal 渲染和衍生的 React Native 项目了。Universal 渲染指的是一套代码可以同时在服务端和客户端渲染，这也是本书重点介绍的技术。React Native 则让你可以使用前沿的 JavaScript 为 iOS、Android 编写跨平台的原生 APP。

如果你对 React 究竟是什么还是很困惑，我的建议是，不用过于纠结，只管先用起来，在实际的使用中慢慢理解它。本书将教你如何用 React 开发 Web 程序。

Redux

Redux 是一个 JavaScript 状态容器，提供可预测的状态管理。

Redux 可以用三条基本原则来描述：单一数据源；state 只读；使用纯函数来执行修改。

单一数据源是指整个应用的 state 被存储在一棵对象树中，并且这个对象树只存在于唯一一个 store 中。这里的 state 指的是数据。单一数据源是 Redux 区别与其他同类工具的一个重要特征。

state 只读并不代表我们无法改变 state。如果无法改变 state，那么程序将不会发生变化。这里“只读”指的是不允许直接对 state 这个变量重写赋值，但是可以通过 action 和 reducer 返回一个新的 state，而且只能使用这一方法。

使用纯函数来执行修改是指更新 state 的 reducer 只是一些纯函数，它接收先前的 state 和 action，并返回新的 state。

了解上述基本原则后，或许你会问：“为什么要使用 Redux?”问得好！使用 Redux 将会给你的项目带来以下这些收益。

- **可预测**：Redux 只有一个数据源，想要修改它只能发起 action，reducer 又是纯函数，相同的输入永远会得到相同的输出。这一切都使得程序运作变得可控、可预测。
- **便于组织管理代码**：严格而明确的程序结构使得代码更容易组织和管理，同时也方便了团队协作。
- **支持 Universal 渲染**：单一数据源这个原则可以帮助解决 Universal 渲染中的数据传递问题，服务端渲染后只需给客户端传递一个变量即可，这个变量就是存储 state 的对象树。

- **优秀的拓展能力**：Redux 支持多种拓展方式，但主要是中间件的拓展。中间件的使用使得原本简单的 Redux 变得十分强大。例如，Redux Thunk 中间件使 Redux 的 action 创建函数具备了异步和条件判断能力，Logger for Redux 中间件则可以为程序添加日志功能。
- **容易测试**：编写可测试的代码，最重要的一条规则就是写单一功能的、没有外界依赖的小型函数。Redux 的代码几乎都是这样小而纯且独立的函数。
- **开发工具**：开发者可以使用 Redux 的开发工具去实时追踪、回退和重放程序中 action 与 state 的每个变化。
- **社区和生态系统**：无论在什么时候，使用什么框架或库，社区和生态系统都是非常重要的。Redux 背后的强大社区使其更加易用。

Reducer 的纯函数、不可变、函数组合，中间件的管道、柯里化等这些思想都属于函数式编程的范畴。在 Redux 中，函数式编程被重度使用，甚至连“Redux”这个名字都充满了浓浓的函数式编程意味（reduce 是函数式编程中经常使用的遍历累加数组元素的方法）。函数式编程使得代码变得更加简洁和模块化。熟悉函数式编程对于理解 Redux 的原理和思想是非常有帮助的。本书将会在介绍 Redux 的同时，穿插讲解函数式编程的知识。希望你在读完本书后能够感受到 Redux 与函数式编程之美。

Node 与 Universal 渲染

React 与 Redux 既可以在浏览器端运行也可以在服务器端运行。这里的服务器端指的是 Node 服务器。

和传统 Web 服务器相比，Node 更简单，它是单线程，与平台无关的。最重要的是，它使用了 JavaScript 这门原来在浏览器中运行的语言，所以我们可以实现 Universal 渲染——用一套代码在服务端和客户端渲染。

最初所有的渲染都是发生在服务器端的，但是在 2005 年，AJAX 的到来将渲染过程转移到了客户端，通过调用服务器的 API 获取和修改数据。此后诞生了无数的 JavaScript 框架来实现客户端渲染和路由控制。但是开发者们意识到一个问题：客户端渲染破坏了搜索引擎的索引，因为搜索引擎无法和 JavaScript 通话。除此之外，客户端渲染在第一次会加载较多模板和脚本，严重影响了网页打开的速度。

很明显，需要同时在服务端和客户端进行渲染，而且最好公用一套代码。于是 Universal 渲染出现了。Universal 渲染实现了更快的加载速度、更好的 SEO、更优雅的交互效果、更容易维护的代码……这一切都关乎你或者你老板的项目利益。作为软件工程师，你应该明白这些，并给予足够的重视，用技术帮助项目盈利。

React 与 Redux 是实现 Universal 渲染的理想技术组合。本书会重点介绍使用 React 与 Redux 实现 Universal 渲染的最佳实践。

Babel

Babel 是一个 JavaScript 编译器，可以让开发者提前使用下一代的 JavaScript。Babel 可以和各种工具配合使用，包括如下这些工具。

- Babel 的内建工具：Babel CLI、Require Hook。
- 各种构建系统：Webpack、Gulp、Grunt、RequireJs 等。
- 测试框架：Jasmine、Karma、Mocha 等。
- 语言 API：C#/.NET、Node、Ruby。
- 模板引擎：Jade。
- 编辑器：WebStorm。
- 调试器：Node Inspector 等。

本书将在 Node 和浏览器两个环境中运行 React 与 Redux 程序，所以我们需要学习在 Node 和 Webpack 中使用 Babel。

Webpack

Webpack 是当下最热门的前端资源模块化管理和打包工具。它可以将许多松散的模块按照依赖和规则打包成符合生产环境部署的前端资源。通过加载器（loader）的转换，任何形式的资源都可以视作模块，比如 CommonJs 模块、AMD 模块、ES2015 模块、CSS、图片、JSON、Coffeescript、LESS 等。

本书使用到的 Webpack 的功能主要包括：

- 编译、加载使用 ES2015 和 JSX 语法的模块。
- 实现开发服务器与热替换。
- 加载图片文件。
- 加载字体文件。
- 加载样式文件。
- 加载 JSON 文件。
- 使用同构工具实现同构渲染。
- 压缩代码。
- 哈希命名。

总结

React 是一个声明式、高效、灵活的、创建用户界面的 JavaScript 库，它所带来的 Universal 渲染更是一场革命。Redux 是一个 JavaScript 状态容器，提供可预测的状态管理，单一数据源、只读 state 等特性使其从同类工具中脱颖而出。而使用 Babel 和 Webpack 做编译和构建则是开发 React 与 Redux 程序的最佳选择。

从下一章开始，我们将循序渐进地学习如何使用这些现代前端技术。

第2章

在Node.js中运行React

本章将介绍如何在Node.js中运行简单的React程序。

例子

本章的例子是一个非常简短的小程序：先编写一个React组件，然后在Node.js中将其渲染成HTML字符串并打印出来。

运行方法：

```
npm install
npm start
```

编写React组件

编写React组件通常需要写一个继承自`React.Component`的类，并在`render()`中返回你要展现的视图：

```
import React from 'react'
export default class App extends React.Component {
  render() {
    return (
      <h1>hello world</h1>
```

```
    )
  }
}
```

但如果这个组件只有一个render()方法，可以将其改为更简洁的无状态函数（stateless function），这也是 Airbnb 编码规范推荐的写法。

src/App.js

```
import React from 'react';

export default function App() {
  return <h1>hello world</h1>;
}
```

在上述代码中，返回<h1>hello world</h1>的无状态函数APP就是我们编写的 React 组件。接下来，我们使用export default默认导出了这个组件。default表示你可以在别的文件中用import App from './App'导入这个模块；如果没有default，则需要用import { App } from './App'。



本例使用了 JSX 语法。JSX 是一个看起来很像 XML 的 JavaScript 语法扩展，这种语法允许你在 JavaScript 中写闭合的标签。也正是因为 JSX 语法的使用，尽管这里表面上没有用到React这个顶层 API，但是仍然需要导入它。在本章我们主要讲解程序的运行，关于 JSX 语法，会在后面的章节详细介绍。

在 Node.js 中渲染组件

编写完 React 组件，我们先在 src/server.js 中导入该组件，然后将其渲染成一个 HTML 字符串并打印出来。在 Node.js 中，应该使用renderToString()方法将组件渲染成字符串。

src/server.js

```
import React from 'react';
import { renderToString } from 'react-dom/server';
import App from './App';

const appHTML = renderToString(<App />);

console.log(appHTML);
```

至此，整个程序的代码就写完了。下面将介绍如何使用 Babel 编译运行这个程序。

使用 Babel 编译运行 Node.js 程序

本章的示例程序采用了 ES2015 语法和 JSX 语法，需要通过 Babel 编译运行。在测试环境中，通常使用 Require Hook 来运行需要 Babel 编译的 Node.js 程序。

Require Hook 简介

Require Hook 是 Babel 的一个内建工具，用于在测试环境下编译运行 Node.js 程序。Require Hook 的工作原理是在 `require` 方法上加上一个钩子。此后，每次使用 `require` 加载 JavaScript 文件时，就会先用 Babel 进行编译。



Bebel 的运行工具很多，可以使用内建的命令行工具，也可以配合各种构建工具，如 Gulp 或 Webpack。如果你之前没有用过 Babel，强烈建议安装使用命令行工具 Babel CLI，体验 Babel 的编译过程。Babel CLI 的用法请参考官网文档（https://babeljs.io/）。

使用 Require Hook 的优势在于将编译和运行这两个步骤合二为一。不过注意，因为性能消耗较大等原因，这个方法只能用于测试环境。如果要在生产环境中运行 Node.js 程序，需要先编译再运行。

使用 Require Hook 的步骤

使用 Require Hook 共四个步骤。

1. 安装 `babel-register`（`npm install`包含了这个安装过程，可以在 package.json 中查看）：

```
npm install babel-register --save-dev
```

2. 在入口文件顶部添加 `require("babel-register")`。

index.js

```
require('babel-register');
require('./src/server.js');
```

3. 安装 ES2015 和 React 的预设：

```
npm install --save-dev babel-preset-es2015 babel-preset-react
```

4. 添加配置文件.babelrc，激活 ES2015 和 React 的预设：

```
{ "presets": ["react", "es2015"] }
```

如此一来，在命令行输入 `npm start`（`npm start` 指向 `node index`，可以在 package.json 中查看），就可以运行这个程序了。



ES2015 和 React 的预设分别用于编译 ES2015 语法和 JSX 语法。作为一个 JavaScript 编译器，Babel 之所以可以按照各种要求编译 JavaScript 代码文件，是因为使用了各种插件和预设（预设是插件的打包封装）。插件或者预设是编译任务的核心“执行者”。如果没有配置插件或者预设，即便运行 Babel，也只是复制了源代码而已。如果你想更多地了解 ES2015 和 React 预设，可以参考 Babel 的官网文档。

总结

- 编写 React 组件通常需要写一个继承自 `React.Component` 的类，并在`render()`中返回你要展现的视图。但如果这个组件只有一个`render()`方法，推荐将其改为更为简洁的无状态函数。
- 在 Node.js 中使用 `renderToString()` 方法将组件渲染成字符串。
- 在测试环境中，通常使用 Require Hook 来运行需要 Babel 编译的 Node.js 程序。

第3章

在浏览器中运行 React

尽管 React 可以跨平台运行，但毫无疑问浏览器仍然是 React 的主战场。本章将会介绍如何在浏览器中渲染 React 组件。

例子

本章例子的功能是在浏览器中渲染 React 组件，运行结果是在浏览器中输出`hello world`。

先安装依赖，然后使用 Webpack 打包编译，最后运行：

```
npm install
npm run build
npm start
```

组件复用

本例中的 App.js 是直接从第 2 章的例子中复制粘贴过来的：

src/App.js

```
import React from 'react';

export default function App() {
```

```
  return <h1>hello world</h1>;
}
```

便于复用是 React 组件一个很大的优点。React 社区非常活跃，有成千上万的开发者在编写 React 组件，我们完全可以从中挑选适合自己的组件，复制粘贴后使用，节省精力。

当然，除了便于复用外，组件化还有很多优点，比如，分开管理代码、减小耦合度等。事实上，不只是 React，任何能撑起大型项目的框架、类库都会提供组件化或模块化的解决方案。

在浏览器中渲染 React 组件

与在 Node.js 中渲染的方法不一样，在浏览器中渲染 React 组件使用了`react-dom`的`render()`方法：

src/client.js

```
import React from 'react';
import { render } from 'react-dom';
import App from './App';

render((<App />), document.querySelector('#app'));
```

以上代码将`<App />`渲染到 ID 为`app`的标签中：

index.html

```
...
<div id="app"></div>
...
```

至此，整个程序的代码就写完了。下面将介绍如何使用 Webpack 将上述代码打包编译成可以在浏览器中运行的脚本文件。

使用 Webpack 打包编译

这个程序使用了 ES2015 语法和 JSX 语法，浏览器目前还不能完全支持，所以需要使用 Webpack 打包编译源代码。

因为 Webpack 本身只有打包功能，所以这里还需要通过引入 babel-loader 来实现编译功能。使用 babel-loader 不是必需的，假如先使用 Babel CLI 编译源代码，再使用 Webpack 打包，就不需要再配置 babel-loader 了。如果你之前没有使用过 Webpack，强烈建议先阅读 Webpack 官网的入门文档（https://webpack.github.io/）。

下面将介绍使用 Webpack 和 babel-loader 打包编译程序的具体步骤。

1. 安装 Webpack 和 babel-loader：

```
npm install --save-dev webpack babel-loader babel-core
```

2. 添加 webpack.config.js 文件来指定打包编译的配置信息：

```
module.exports = {
  entry: './src/client',
  output: {
    path: __dirname + '/static/dist',
    filename: 'main.js'
  },
  module: {
    loaders: [{ test: /\.js$/, exclude: /node_modules/, loaders: ['babel'] }]
  }
};
```

entry表示入口文件，output中的path表示输出目录，filename表示输出的文件名称。只有下面这部分配置信息是用于 babel-loader 的：

```
module: {
    loaders: [{ test: /\.js$/, exclude: /node_modules/, loaders: ['babel'] }]
  }
```

它的意思是除了/node_modules/目录下的文件之外，其他所有符合/\.js$/正则表达式的文件都将使用 babel-loader 编译。

3. 安装 ES2015 和 React 的预设：

```
npm install --save-dev babel-preset-es2015 babel-preset-react
```

4. 添加配置文件.babelrc：

```
{
    "presets": ["react", "es2015"]
}
```

现在在目录下运行npm run build命令（npm run build指向webpack）来打包和编译，结果成功生成了可以在浏览器中运行的脚本文件，它的路径为 static/dist/main.js。

在浏览器中运行

打包编译工作完成后，我们将生成的 JavaScript 脚本文件添加到 index.html 中。

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>React Demo</title>
</head>

<body>
    <div id="app"></div>
    <script src="static/dist/main.js"></script>
</body>

</html>
```

然后在目录下运行npm start命令（npm start命令指向http-server）。此时，在这个目录下已经启动了一个服务，在浏览器中打开这个服务的路径就可以运行 index.html 了。

总结

- 一个 React 组件既可以在 Node.js 中渲染，也可以在浏览器中渲染。
- 渲染组件到 DOM 节点中时使用了react-dom的render()方法。
- 浏览器目前无法直接运行用 ES2015 和 JSX 语法编写的 JavaScript 脚本，需要使用 Webpack 和 babel-loader 打包编译。

第4章

开发服务器和热替换

在第 3 章的示例程序中，每次修改代码后，都必须执行一次npm run build命令进行编译，然后再刷新浏览器，开发调试效率非常低下。为了避免这种低效的重复工作，本章的例子使用了开发服务器（webpack-dev-server）和热替换（hot reloading）技术。开发服务器能够在指定端口提供资源服务，配合热替换则可以自动监视代码的变动，重载被修改的模块，不仅不需要刷新浏览器，甚至连程序原来的状态都可以被保留下来（刷新浏览器会重置程序状态），极大地方便了开发调试。下面让我们来学习使用这个技术。

注意，本章涉及的概念很多，包括 Babel、Webpack、源代码映射和 Express 中间件等，对于初学者而言，很难在短时间内掌握。如果对本章内容感到吃力，请不要细究每个知识的内部原理，先学习使用方法。随着日后开发经验的增长，之前有困惑的知识原理慢慢就会理解了。

例子

本章例子使用了开发服务器和热替换在浏览器中运行 React 程序，尝试改变 React 组件的代码，看看会发生什么。

运行方法：

```
npm install
npm start
```

安装

npm install命令已经安装了 package.json 中dependencies和devDependencies选项下的所有 NPM 包，所以这里无须重复安装。但是在你自己项目中，假设已经安装了 React 和 Express，那么你还需要安装这些包：

```
npm install --save-dev babel-core babel-loader babel-preset-es2015 babel-preset-react
    babel-preset-react-hmre webpack webpack-dev-middleware webpack-hot-middleware
```

在以上这些包中，有很大一部分都已经在前面的章节中接触过了。下面几节将会介绍如何配置 Babel、Webpack 和 Express 服务器，并在这个过程中讲解其他包的功能与用法。

配置 Babel

热替换是指当修改部分代码后，程序会自动编译、替换被修改的代码，不需要重新编译所有代码，也不需要刷新浏览器。既然热替换涉及编译，所以首先应该配置 Babel。下面将介绍配置 Babel 的具体细节。

使用 react-hmre 预设

配置 Babel 使其支持热替换的最快捷的方法是使用react-hmre预设：

.babelrc

```
{
  "presets": ["es2015", "react"],
  "env": {
    "development": {
      "presets": ["react-hmre"]
    }
  }
}
```

上述配置在 ES2015 和 React 预设的基础上，增加了react-hmre这个预设，并且限制只在开发环境下使用。



注意，这里并不一定非要使用react-hmre这个预设，也可以自己配置。事实上，Babel 的预设也只是对配置的打包封装而已。如果你想了解react-hmre这个预设的具体细节，可以查看它的 GitHub 仓库（https://github.com/danmartinez101/babel-preset-react-hmre）。

react-hmre 的功能

总体来说，react-hmre这个预设主要包含两个功能：热替换 React 模块和捕捉错误。

热替换 React 模块

react-hmre这个预设的第一个功能是热替换 React 模块。如果没有这个预设，在代码变动时，将无法正确地重新载入 React 模块。尝试删除这个预设的配置信息：

```
{
  "presets": ["es2015", "react"],
-  "env": {
-    "development": {
-      "presets": ["react-hmre"]
-    }
-  }
}
```

重新npm start，然后修改你的 React 组件，比如加个console.log()，结果在浏览器控制台里出现了警告信息。

捕捉错误

另外，react-hmre这个预设还可以捕捉错误，并将包含错误堆栈信息的红色警告页面输出到浏览器中。

```
export default class App extends Component {
  render() {
    throw new Error();
    return <h1>hello world</h1>;
  }
}
```



注意，如果将无状态函数放在组件顶层，热替换将会出错。所以本章示例程序将 src/App.js 中的无状态函数改成了类组件。

尝试在组件中抛出一个错误对象，结果浏览器中输出了警告页面，如图 4-1 所示。

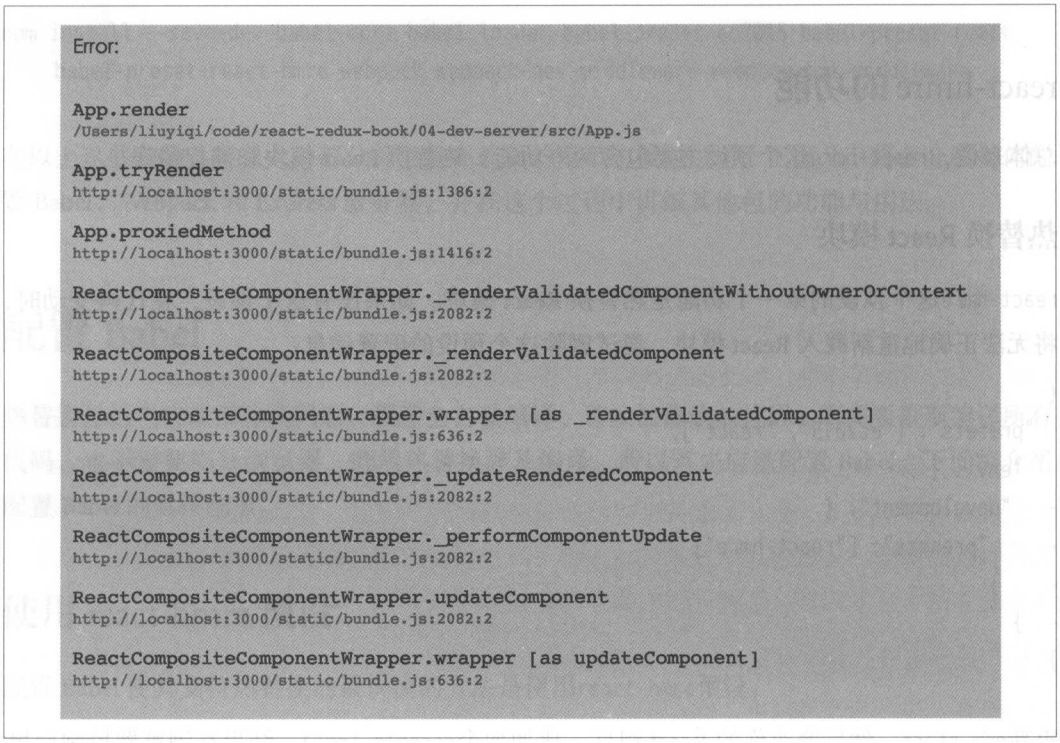


图 4-1

配置 Webpack

开发服务器是可以为程序提供资源服务的服务器。通常情况下，在页面中引入的脚本文件等静态资源都放在硬盘中。但是，使用了开发服务器，这些资源将被读到内存里，然后我们可以通过开发服务器的端口来访问这些资源。开发服务器结合热替换可以实现监视代码变动，即时更新界面的功能。因为开发服务器是通过 Webpack 实现的，因此在搭建服务器前需要先配置 Webpack：

webpack.config.js

```
var path = require('path');
var webpack = require('webpack');

module.exports = {
  devtool: 'cheap-module-eval-source-map',
  entry: [
    'webpack-hot-middleware/client',
    './index.js'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: '/static/'
  },
  plugins: [
    new webpack.optimize.OccurrenceOrderPlugin(),
    new webpack.HotModuleReplacementPlugin()
  ],
  module: {
    loaders: [
      {
        test: /\.js$/,
        loaders: ['babel'],
        exclude: /node_modules/,
        include: __dirname
      }
    ]
  }
};
```

这个 Webpack 配置文件的选项解释如下。

- devtool是开发工具，它的作用是生成源代码映射（Source Map），方便调试。

 源代码映射记录了打包编译后的代码和源代码之间的位置对应关系。它可以让你在控制台查找日志（或错误）来源时，直接跳转到源代码中，而不是打包编译后的代码。

如果你第一次接触源代码映射，可以在代码中打印一段信息或抛出错误，然后打开浏览器的开发者工具，点击日志右边的链接（例如图 4-2 中的App.js?14b1:8）试试。

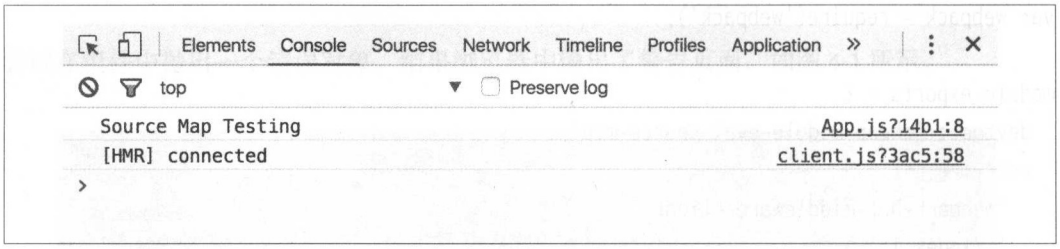


图 4-2

开发工具有若干个模式可以选择，它们的区别主要在于构建速度和源代码映射的质量。这里选择cheap-module-eval-source-map。这个模式构建速度适中，而且会生成较好的源代码映射，适合在开发环境中使用。更多开发工具模式的介绍，请查阅 Webpack 官网文档。

- entry是入口文件，本例的入口选项包括：
 - 程序的入口文件./index.js。
 - 用于热替换功能的webpack-hot-middleware/client。webpack-hot-middleware/client 的工作原理是：先连接上服务器，等着接受需要重新编译代码的通知，一旦代码变动，就会收到通知进而更新客户端代码。
- output是输出配置。下面是输出选项中各个子选项的含义：
 - path表示输出的硬盘路径，不过本例中编译好的代码放在内存中，故此项无用。
 - filename表示输出的文件名称。
 - publicPath则表示公共路径，即所有资源的 URL 前缀。比如，如果你的 HTML 页面中有一个<script src="/static/bundle.js"></script>，那么应该将publicPath设为/static/。在本例中，这个路径是虚拟的（即无法在硬盘中找到这个路径），打包编译后的脚本被保存到了内存中，可以通过localhost:3000/static/bundle.js访问。publicPath选项的用法还有很多，这些我们会在后续章节详细介绍。
- plugins是插件列表。Webpack 中的插件被用于实现与打包相关的特定功能。Webpack 提供了很多内置的插件可以直接使用。除此之外，也可以通过安装 NPM 包来使用一些第三方插件。本例使用了两个内置插件来实现热替换，它们的功能解释如下：

 - webpack.optimize.OccurrenceOrderPlugin用于给经常使用的模块分配最小长度的ID。
 - webpack.HotModuleReplacementPlugin则用于热替换。

 想了解更多的 Webpack 插件，可以参考 Webpack 的官网文档。

- module.loaders是加载器列表，此项配置与前面章节的例子中相同，不再赘述。

至此，Webpack 的配置就完成了。

配置 Express 服务器

在配置完 Webpack 后，只需要在 Express 服务器中添加 webpackDevMiddleware 和 webpackHotMiddleware 两个中间件即可完成开发服务器和热替换的配置工作。

在 Express 中，中间件是服务器在收到请求（Request）和做出响应（Response）之间的一个处理器。Express 中间件可以多个串联使用，它是 Express 框架最主要的一种拓展方式，我们可以使用中间件完成诸如记录日志、内容压缩等工作。现将 webpackDevMiddleware 和 webpackHotMiddleware 这两个中间件的功能介绍如下：

- webpackDevMiddleware：Webpack 本身只有打包功能，但是把这个中间件连入 Express 服务器后，就可以将 Webpack 的打包功能与 Express 服务器的资源服务功能合二为一。Express 服务器通过该中间件完成打包工作，并将打包好的资源读到内存中，接下来我们就可以通过 Express 服务器的端口来访问这些资源了。不仅如此，该中间件该可以监视代码变动，一旦代码变动，就会停止提供旧版的资源服务，等到编译完成后继续提供新的资源服务。
- webpackHotMiddleware：webpackDevMiddleware中间件已经完成了大部分自动化工作，但是它无法实现热替换，也就是无法在更新代码后保留程序状态。为了实现热替换，还应该在webpackDevMiddleware中间件的基础上加上webpackHotMiddleware中间件。

server.js

```
/* eslint-disable */
var webpack = require('webpack');
var webpackDevMiddleware = require('webpack-dev-middleware');
var webpackHotMiddleware = require('webpack-hot-middleware');
var config = require('./webpack.config');
```

```
var app = new (require('express'))();
var port = 3000;

var compiler = webpack(config);
app.use(webpackDevMiddleware(compiler, { noInfo: true, publicPath: config.output.
    publicPath }));
app.use(webpackHotMiddleware(compiler));

app.get("/", function(req, res) {
  res.sendFile(__dirname + '/index.html')
});

app.listen(port, function(error) {
  if (error) {
    console.error(error)
  } else {
    console.info("==> 🌎  Listening on port %s. Open up http://localhost:%s/ in your
        browser.", port, port)
  }
});
```

配置完成，启动服务，打开浏览器修改代码，将会看到浏览器自动重载了变化的模块。除了观察界面上的变化之外，还可以打开控制台面板，查看代码重建的详细信息。关于webpack-DevMiddleware 和webpackHotMiddleware这两个中间件更多的配置使用方法，请参考它们的官网文档。

至此，使用 Webpack 和 Babel 进行编译构建的基础知识就讲完了。之所以将这些知识放在前面讲解，是因为它们是全书示例程序的基础。从下一章开始，我们将会讲解 React 的一些相关知识。

总结

- 使用开发服务器和热替换需要配置 Babel、Webpack、Express。
- Babel 的配置主要是在 ES2015 和 React 预设的基础上，使用了react-hmre这个预设。

- Webpack 的配置包括：使用webpack-hot-middleware/client作为入口文件，包含 new webpack.optimize.OccurrenceOrderPlugin 和 new webpack.HotModuleReplacementPlugin() 插件等。
- Express 的配置则是使用了webpackDevMiddleware和webpackHotMiddleware这两个中间件。

第5章

React的创新语法：JSX

React之所以可以声明式地快速构建用户界面，JSX语法绝对功不可没。本章将介绍一些常用的JSX语法，并给出示例。这些知识是编写React组件的基础，需要熟练掌握。

例子

本章例子编写了一些小组件来演示JSX常用的语法规则（见图5-1）。

JSX简介

JSX是一个看起来很像XML的JavaScript语法扩展。这种语法允许你在JavaScript中写可嵌套的闭合标签。

JSX并不复杂，它只是一个语法糖。使用JSX编写的代码最终都会被转换成普通的JavaScript代码。因此，也可以不用JSX去编写React程序。但是最好不要这么做，因为相比于普通JavaScript的写法，JSX可以让组件的结构和组件之间的关系看上去更加清晰。

下面几节将通过一些代码片段讲解JSX的语法细节。

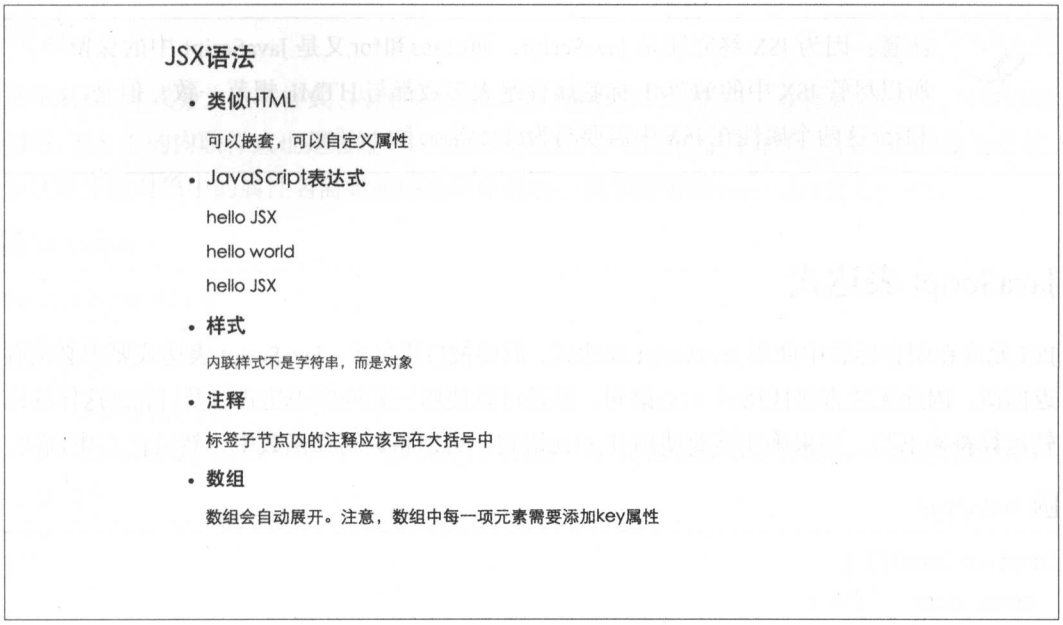


图 5-1

JSX 常用语法

类似 HTML

JSX 与 HTML 非常相似，可以嵌套多个 HTML 标签，也可以使用大部分符合 HTML 规范的属性，比如`style`。如果往 HTML 标签里传入了 HTML 规范里不存在的属性，React 不会显示它们，但是可以通过加上`data-`前缀的办法自定义属性。

src/App.js

```
function Demo1() {
  return (
    <li>
      <h3>类似HTML</h3>
      <p data-attribute="demo1">可以嵌套，可以自定义属性</p>
    </li>
  );
}
```



注意：因为 JSX 终究还是 JavaScript，而class和for又是 JavaScript 中的保留字，所以尽管 JSX 中的 HTML 标签属性绝大多数都与 HTML 规范一致，但是class和for这两个属性在 JSX 中需要写为className和htmlFor。

JavaScript 表达式

JSX 允许在闭合标签中使用 JavaScript 表达式，但要被{}所包裹。JavaScript 表达式要求必须有返回值，因此无法直接使用if else语句，但是可以使用三元操作表达式以及||和&&这样的比较运算符来书写。如果确实需要使用if else语句，可以将其写在函数中，然后在{}中调用。

src/App.js

```
function Demo2() {
  const name = 'JSX';
  const func = () => {
    let result = 'hello ';
    if (name) {
      result += name;
    } else {
      result += 'world';
    }
    return result;
  };
  return (
    <li>
      <h3>JavaScript表达式</h3>
      <p>hello {name || 'world'}</p>
      <p className={name ? 'class-a' : 'class-b'}>
        hello {name && 'world'}
      </p>
      <p>
        {func()}
      </p>
    </li>
  );
}
```

样式

总体来说，样式分为内联样式、内嵌式、链接式等，这里要讲解的是内联样式的写法。与 HTML 相似，JSX 中的内联样式也是通过style属性来定义的，但属性值不能是字符串而必须为对象，而且要注意对象中的属性名需要使用驼峰命名法，例如需要将font-size变为fontSize。

src/App.js

```
function Demo3() {
  return (
    <li>
      <h3>样式</h3>
      <p style={{ color: 'red', fontSize: '14px' }}>内联样式不是字符串，而是对象</p>
    </li>
  );
}
```

注释

在 JSX 中添加注释非常简单，只需要注意将标签子节点内的注释写在{}中就可以了。

src/App.js

```
function Demo4() {
  return (
    <li>
      <h3>注释</h3>
      {/* 注释... */}
      <p>标签子节点内注释应该写在大括号中</p>
    </li>
  );
}
```

数组

JSX 中的数组会自动展开所有成员。但是需要注意，如果数组或迭代器中的每一项都是 HTML 标签或组件，那么它们必须要拥有唯一的key属性。这样做是为 React 的 DIFF 算法服务的，React 会通过唯一的key属性实现最高效的 DOM 更新。

src/App.js

```
function Demo5() {
  const arr = [
    <h3 key={0}>数组</h3>,
    <p key={1}>数组会自动展开。注意，数组中每一项元素需要添加key属性</p>,
  ];
  return (<li>{arr}</li>);
}
```

HTML 标签 vs. React 组件

React 可以渲染 HTML 标签或 React 组件。HTML 标签使用小写字母的标签名，而 React 组件的标签名首字母则需要大写。

src/App.js

```
export default class App extends Component { // eslint-disable-line
  render() {
    return (
      <div>
        <h2>JSX语法</h2>
        <ul>
          <Demo1 />
          <Demo2 />
          <Demo3 />
          <Demo4 />
          <Demo5 />
        </ul>
        ...
      </div>
    );
  }
}
```

至此，常用的 JSX 的语法以及相关知识就讲完了。如果想了解更多的 JSX 内容，可以查阅官网文档（https://facebook.github.io/jsx/）。

总结

- JSX 是一个看起来很像 XML 的 JavaScript 语法扩展。这种语法允许你在 JavaScript 中写可嵌套的闭合标签。
- JSX 与 HTML 语法很像，可以嵌套，可以自定义属性。
- JSX 允许在闭合标签中使用 JavaScript 表达式，但要被{}所包裹。
- JSX 中的内联样式也是通过`style`属性来定义的，但属性值不能是字符串而必须为对象，而且要注意对象中的属性名需要使用驼峰命名法。
- 在 JSX 中，标签子节点内的注释应该写在大括号中。
- JSX 中的数组会自动展开所有成员。但是需要注意，如果数组或迭代器中的每一项都是 HTML 标签或组件，那么它们必须要拥有唯一的`key`属性。

第6章

React 的数据载体：state、props 与 context

本章要讲解的是 React 中三个非常重要的概念：state、props 与 context。State 其实应该被称为内部状态或局部状态（local state）。“内部”表示它很少“跑”出组件，“状态”则意味着它经常变化。Props 与 context 则用于在组件间传递数据，props 仅支持逐层传递，而 context 则能够跨级传递。State、props 与 context 都是 React 中的数据载体，它们各司其职，让数据在组件中优雅地变化和流动。

例子

本章例子编写了三个组件：

- 第一个组件是一个计数器，演示了如何使用内部状态和 props 去更新传递数据。
- 第二个组件是一个消息列表，演示了如何通过 props 将数据逐层传递给最里面的组件。
- 第三个组件也是一个消息列表，演示了如何通过 context 将数据跨级传递给最里面的组件。

运行结果如图 6-1 所示。



图 6-1

State

严格来说，state 应该被称为内部状态或局部状态（local state）。它是 React 组件中一个较为常用的概念。React 组件可以在构造函数中初始化内部状态，可以通过this.setState方法更新内部状态，还可以使用this.state获取内部状态。这些内部状态的操作与 React 的事件系统配合就可以实现一些用户交互的功能。

src/Counter.js

```
export default class Counter extends Component {
  constructor() {
    super();
```

```
    this.state = { value: 0 };
  }

  render() {
    return (
      <div>
        <button onClick={() => this.setState({ value: this.state.value + 1 })}>
          INCREMENT
        </button>
        Counter组件的内部状态：
        <pre>{JSON.stringify(this.state, null, 2)}</pre>
      </div>
    );
  }
}
```

上述代码首先在构造函数中初始化内部状态，然后给button的点击事件注册了更新内部状态的方法，最后将内部状态序列化显示在pre标签中。



constructor是ES2015类中的构造函数，它仅在实例化一个类的时候被调用。另外，在子类的构造函数中，必须先调用super()，才能使用this获取实例化对象。这是ES2015类的一个规则。constructor是类的默认方法，通常无须编写。但是如果想在构造函数中编写一些逻辑（比如，初始化内部状态、为组件方法绑定this等），就需要单独编写了。

随着无状态函数（无状态函数没有内部状态）的提出和Redux的使用，内部状态的使用正在逐渐减少。但是内部状态在非全局的数据管理更新中仍扮演着重要的角色。

Props

Props是React中非常重要的一个概念。顾名思义，props就是属性的意思。我们可以使用props向React组件传递数据，React组件从props中拿到数据，然后返回视图。

使用 props

向一个组件传递 props 的方法是将数据写在组件标签的属性中。

src/Counter.js

```
<Content value={this.state.value}/>
```

那么组件如何获取 props 呢？在无状态函数编写的组件中获取 props 非常简单，只需要将 props 作为参数传入组件即可。

src/Counter.js

```
function Content(props) {
  return <p>Content组件的props.value：{props.value}</p>;
}
```

在使用类编写的组件中，需要通过this.props获取 props。this是组件实例。关于组件实例，我们会在后面的章节详细讲解。

验证 props

在一个健壮的程序中，任何输入都是需要验证的，组件也不例外。Props 作为组件的输入，必须进行验证。验证 props 需要使用React.propTypes，它提供很多验证器（validator）来验证传入数据是否合法。当向 props 传入非法数据时，控制台会抛出警告。

PropTypes提供了很多了验证器，列举如下。

1. JavaScript 基本数据类型，包括数组、布尔、函数、数字、对象、字符串：

   ```
   React.PropTypes.array
   React.PropTypes.bool
   React.PropTypes.func
   React.PropTypes.number
   React.PropTypes.object
   React.PropTypes.string
   ```

2. 可以被渲染为子节点的对象，包括数值、字符串、ReactElement（指的是 JSX 中的闭合标签）或数组：

   ```
   React.PropTypes.node
   ```

3. ReactElement：

```
React.PropTypes.element
```

4. 指定类的实例：

```
React.PropTypes.instanceOf(Message)
```

5. 只接受指定的值：

```
React.PropTypes.oneOf(['News', 'Photos'])
```

6. 多个对象类型中的一个：

```
React.PropTypes.oneOfType([
      React.PropTypes.string,
      React.PropTypes.number,
      React.PropTypes.instanceOf(Message)
    ])
```

7. 指定类型组成的数组：

```
React.PropTypes.arrayOf(React.PropTypes.number)
```

8. 指定类型的属性构成的对象：

```
React.PropTypes.objectOf(React.PropTypes.number)
```

9. 符合指定格式的对象：

```
React.PropTypes.shape({
      color: React.PropTypes.string,
      fontSize: React.PropTypes.number
    })
```

10. 任意类型加上 isRequired 使其不可空：

```
React.PropTypes.func.isRequired
```

11. 自定义验证器。如果验证失败需要返回一个 Error 对象。不要直接使用 console.warn 或抛异常，因为这样的话 oneOfType 会失效。

```
function(props, propName, componentName) {
      if (!/matchme/.test(props[propName])) {
        return new Error('Validation failed!');
      }
    }
```

本章的例子使用了PropTypes.number.isRequired来保证Content组件的props.value必须为数字类型，且不能为空。

src/Counter.js

```
import React, { Component, PropTypes } from 'react';

...

Content.propTypes = {
  value: PropTypes.number.isRequired
};
```

组合使用 state 与 props

下面将演示如何在应用中组合使用 state 和 props。这里使用了一个Counter组件更新state.value，然后将更新的state.value通过 props 传递给Content组件，最后Content组件在每次更新时都渲染出新接收到的props.value。

src/Counter.js

```
import React, { Component, PropTypes } from 'react';

function Content(props) {
  return <p>Content组件的props.value：{props.value}</p>;
}

Content.propTypes = {
  value: PropTypes.number.isRequired
};

export default class Counter extends Component {
  constructor() {
    super();
    this.state = { value: 0 };
  }

  render() {
    return (
```

```
      <div>
        <button onClick={() => this.setState({ value: this.state.value + 1 })}>
          INCREMENT
        </button>
        Counter组件的内部状态：
        <pre>{JSON.stringify(this.state, null, 2)}</pre>
        <Content value={this.state.value}/>
      </div>
    );
  }
}
```

Context

Context 在 React 中是个比较不常用的概念。但是因为后续章节的 react-redux 会用到 context，所以我们将其拿出来和 props 对比讲解一下。下面几节将会通过两个例子来分别演示使用 props 和 context 传递数据的过程，最后总结它们的适用场景。

使用 props 传递数据

为了演示如何通过 props 向里面的组件传递数据，我们编写了三个组件，它们分别是按钮（Button）、消息（Message）和消息列表（MessageList）。接下来，我们会在最外层的消息列表组件中定义一个color变量，然后通过 props 将color传给最里面的按钮组件。

src/Messagelist1.js

```
import React, { PropTypes } from 'react';

function Button(props) {
  return (
    <button style={{ background: props.color }}>
      {props.children}
    </button>
  );
}
```

```
Button.propTypes = {
  color: PropTypes.string.isRequired,
  children: PropTypes.string.isRequired
};

function Message(props) {
  return (
    <li>
      {props.text} <Button color={props.color}>Delete</Button>
    </li>
  );
}

Message.propTypes = {
  text: PropTypes.string.isRequired,
  color: PropTypes.string.isRequired
};

function MessageList() {
  const color = 'gray';
  const messages = [
    { text: 'Hello React' },
    { text: 'Hello Redux' }
  ];
  const children = messages.map((message, key) =>
    <Message key={key} text={message.text} color={color}/>
  );
  return (
    <div>
      <p>通过props将color逐层传递给里面的Button组件</p>
      <ul>
        {children}
      </ul>
    </div>);
}

export default MessageList;
```

上述代码做了几件事：

- 在消息列表组件中遍历生成消息组件，并向其传递了key、text、color等 props。虽然key这个 props 没有用到，但是当你遍历生成一个组件列表时，仍然需要给列表中的每一项传递key。这是因为 React 的 DIFF 算法需要key来做重新排列，以高效地更新 DOM。另外，因为消息组件没有子元素，所以应该写为自闭合的形式，如<XXX />，这也是 Airbnb 编码规范推荐的写法。
- 在消息组件中，渲染按钮组件，并向其传递了color。
- 在按钮组件中，渲染了一个普通按钮，并声明式地定义它的背景颜色为上级组件传来的color。

消息组件并没有使用color，但是为了给按钮组件传递color，不得不从消息列表中接受color。本章例子只有三层组件，假如你的程序中有更深层次的组件结构，你一定不愿意逐层传递 props。此时，context 将帮助你解决这个问题。

使用 context 传递数据

与使用 props 传递数据相比，使用 context 传递数据只需要两步即可实现跨级传递。第一步，将要传递的数据放在消息列表组件的 context 中。第二步，在按钮组件中声明contextTypes，就可以通过组件实例的context属性访问接收到的数据了。事实上，只要在一个组件中定义了 context，这个组件里面的子组件，不管跨多少级，都可以访问到 context 中的数据。

src/Messagelist2.js

```
import React, { Component, PropTypes } from 'react';

function Button(props, context) {
  return (
    <button style={{ background: context.color }}>
      {props.children}
    </button>
  );
}

Button.propTypes = {
```

```
  children: PropTypes.string.isRequired
};

Button.contextTypes = {
  color: PropTypes.string.isRequired
};

function Message(props) {
  return (
    <li>
      {props.text} <Button>Delete</Button>
    </li>
  );
}

Message.propTypes = {
  text: PropTypes.string.isRequired
};

class MessageList extends Component {
  getChildContext() {
    return { color: 'gray' };
  }

  render() {
    const messages = [
      { text: 'Hello React' },
      { text: 'Hello Redux' }
    ];
    const children = messages.map((message, key) =>
      <Message key={key} text={message.text}/>
    );
    return (
      <div>
        <p>通过context将color跨级传递给里面的Button组件</p>
        <ul>
          {children}
        </ul>
```

```
      </div>
    );
  }
}

MessageList.childContextTypes = {
  color: PropTypes.string.isRequired
};

export default MessageList;
```

上述代码做了几件事：

- 在消息列表组件中，遍历生成消息组件，并向其传递key和text这两个 props。然后通过getChildContext()方法将color放在 context 中，并声明了childContextTypes。注意，如果不声明childContextTypes，将无法在组件中使用getChildContext()方法。
- 在消息组件中，渲染按钮组件，这里不用再直接传递color了。
- 在按钮组件中，首先传递父组件的 props 和全局的 context，然后渲染一个普通按钮，并声明式地定义它的背景颜色为context.color。注意，如果没有定义contextTypes，context将会是个空对象。

另外，无状态组件可以直接在函数参数中获取 context，而有状态组件可以通过this.context和生命周期函数获取 context。

Props 与 context 的适用场景

React 的 context 和全局变量非常相似，在大多数场景下，我们都应该尽量避免使用。适合使用 context 的场景包括传递登录信息、当前语言以及主题信息等。另外，react-redux 的 Provider 组件就使用了 context 来传递 store，我们后续会详细讲解。

如果只是传递一些功能模块的数据，则尽量不要使用 context，使用 props 传递数据会更加清晰和容易理解；而使用 context 则会使你的组件的复用性降低，因为这些组件依赖“上下文”，当你在别的地方渲染它们时，可能会出现一些差异。

在 React 开发者工具中查看

讲了这么多概念，让我们在 React 开发者工具中看看这些 state、props 和 context 吧！

React 开发者工具是个 Chrome 插件，你可以通过在 Chrome 网上应用店里面搜索“React Developer Tools”找到它。

打开 Chrome 的开发者工具，你会看到多了一个 React 面板。点击这个面板，就能看到里面包含了当前程序的所有组件。点击这些组件，可以在右侧看到名称、props、state、context 等 React 组件的常用属性，同时浏览器中与此组件对应的界面也会高亮显示。除此之外，React 开发者工具还有追踪更新、查找组件等实用功能。

让我们运行本章的例子，并使用 React 开发者工具来查看这个程序。

可以在Counter组件中看到内部状态State，如图 6-2 所示。

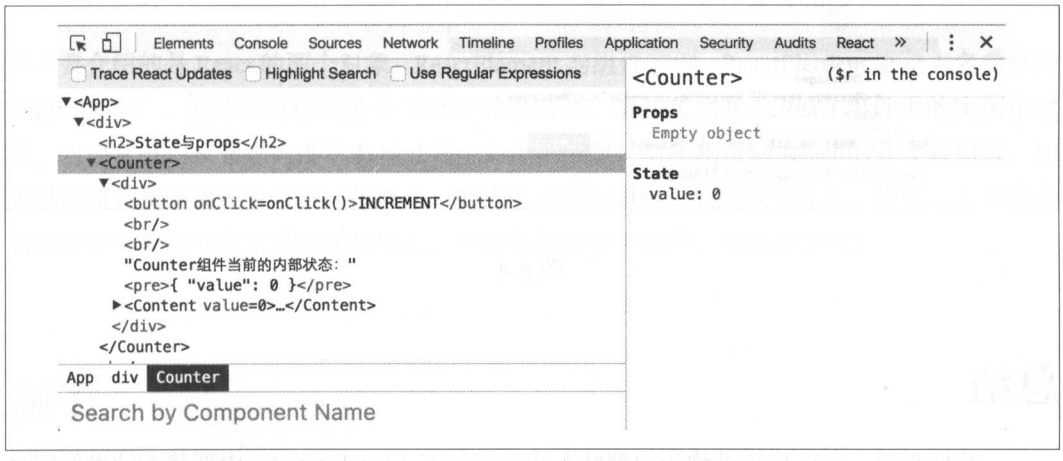


图 6-2

可以在通过 props 传递数据的的Button组件中看到 props，如图 6-3 所示。

还可以在通过 context 传递数据的Button组件中看到 context，如图 6-4 所示。



图 6-3



图 6-4

总结

- 严格来说，state 应该被称为内部状态或局部状态（local state）。内部状态的操作配合 React 的事件系统可以实现一些用户交互的功能。
- Props 与 context 则用于在组件间传递数据。使用 props 传递数据简单清晰，但是跨级传递非常麻烦。使用 context 可以跨级传递数据，但是会降低组件的复用性，因为这些组件依赖“上下文”。所以尽量只使用 context 传递登录状态、颜色主题等全局数据。
- 在 React 开发者工具中，可以清晰地看到每个组件的 state、props 以及 context 等信息，便于开发和调试。

第7章

React的两个对象：ReactElement与组件实例

本章要介绍的是React的两个对象：ReactElement与组件实例。ReactElement就是大名鼎鼎的“虚拟DOM”，但其本质也只是个不可变对象而已。组件实例就是React组件类的实例化对象，它通常被用来管理内部状态和处理生命周期函数。对于刚接触React的开发者而言，如果只是通过文字描述去了解这些概念，恐怕很难完全明白它们究竟是什么。因此，本章将会通过动手实验的形式来讲解这些概念。下面就打开示例程序，边做边学吧！

例子

本章例子首先通过打印组件、ReactElement和组件实例的方式对比讲解它们的相关知识，然后在页面中添加了三个按钮，通过点击按钮，去挂载、更新、卸载一个React组件，并在以此探究组件生命周期函数的触发时机以及React组件方法中的`this`指向问题（见图7-1）。

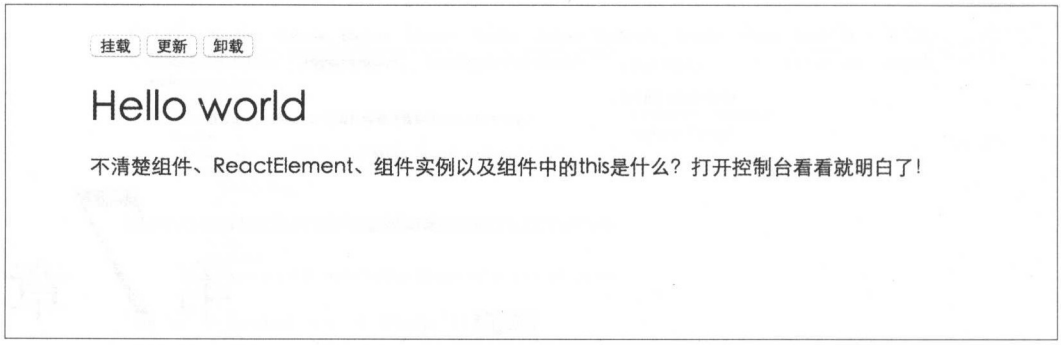


图 7-1

ReactElement

JSX 中的闭合标签是 ReactElement

JSX 允许在 JavaScript 中写可嵌套的闭合标签。如果你想知道 JSX 中的闭合标签究竟是什么，可以把它打印出来看看：

index.js

```
console.log(<h1>hello world</h1>);
console.log(<App />);
```

运行程序后，可以在浏览器控制台中看到打印了两个$$typeof为Symbol(react.element)的对象，这个对象叫 ReactElement（见图 7-2）。



注意，只有在 React 中使用 JSX，闭合标签才会被编译为 ReactElement。假如在其他框架或库中（比如 Vue）使用 JSX，闭合标签将会根据编译器的不同而被编译成不同的对象。

ReactElement 是什么

ReactElement 是一个不可变的普通对象，它描述了一个组件的实例或一个 DOM 节点。它只包含组件的类型（比如h1、或者APP）、属性以及子元素等信息。

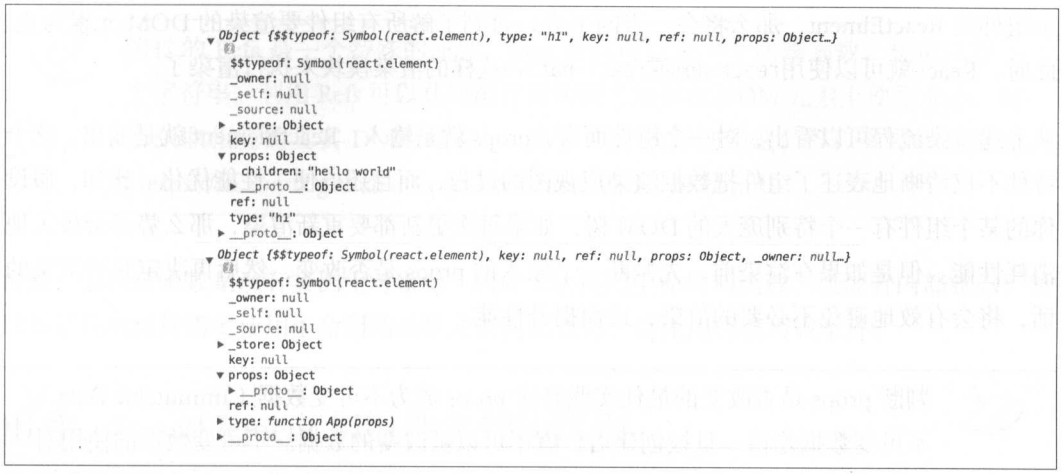


图 7-2

ReactElement 不是组件的实例，不能在 ReactElement 中调用 React 组件的任何方法。它只是告诉 React 你想在屏幕上显示什么。

ReactElement 的两种类型

从图 7-2 可以看到，两个 ReactElemnt 对象的type属性有所不同。这正代表了 ReactElement 的两种类型。

当 ReactElement 的type属性是一个字符串时，它表示了一个 DOM 节点，它的props属性对应了 DOM 节点的属性。图 7-2 中的第一个 ReactElement 对象就是这种类型。

当 ReactElement 的type属性是一个表示组件的函数或类时，它表示了一个组件，就像图 7-2 中的第二个 ReactElement 对象那样。

上述两种类型的 ReactElement 可以相互嵌套，彼此混合来描述 DOM 树。

React 组件的渲染流程

了解了 ReactElement 的两种类型，接下来介绍一下 React 组件的渲染流程。

当 React 遇到表示组件的 ReactElment 时，它会给这个 ReactElement 表示的组件一些 props（有时候也包括 context），然后问该组件渲染的 ReactElement 是什么。如果渲染的仍然是表

示组件的 ReactElment，那么将会一直问下去，直到了解所有组件要渲染的 DOM 元素为止。此时，React 就可以使用`react-dom`或`react-native`这样的渲染模块来执行渲染了。

从上述渲染流程可以看出，对一个组件而言，props 就是输入，ReactElement 就是输出。这个特性不仅清晰地表述了组件把数据渲染成视图的过程，而且还方便了性能优化。比如，假设你的某个组件有一个特别庞大的 DOM 树，如果每次更新都要重新渲染，那么势必会极大地消耗性能。但是如果在渲染前，先判断一下输入的 props 是否改变，然后再决定是否渲染的话，将会有效地避免不必要的渲染，进而提升性能。



判断 props 是否改变的最佳实践是让 props 成为不可变数据（Immutable Data）。不可变数据是指一旦被创建后，就不可以被改变的数据。不可变数据的使用对于实现回退、处理数据变化检查等工作都大有裨益。如果 props 是不可变数据的话，那么判断 props 是否改变就变得容易多了，而且速度很快。这也是为何 React 与不可变数据配合良好的原因。在本书中，我们使用了 Redux 来管理状态，Redux 的 state 正是不可变数据。关于 Redux 和不可变数据的相关知识，我们会在后续章节详细介绍。

ReactElement 在 React 中非常重要。在编写 React 组件时，开发者大部分时间都会花在设计编写 ReactElment 所组成的元素树上面。与 ReactElement 相比，组件实例这个概念就陌生多了。下面我们就来了解一下组件实例的相关知识。

组件实例

组件实例简介

组件实例是组件类的实例化对象，它通常被用来管理内部状态、处理生命周期函数。大多数情况下，我们无需直接创建组件实例，React 会负责创建它。`ReactDOM.render`方法返回的就是组件实例。除此之外，某些组件方法中的`this`也指向组件实例。利用组件的 Refs 也可以获取组件实例。



组件的 Refs 是一个特殊的属性，Refs 属性可以是一个回调函数，也可以是一个字符串。利用 Refs 可以获取组件的实例（如果在 DOM 元素上使用 Refs，将会获取相应的真实 DOM 节点）。关于 Refs 的详细用法，可以参考官网文档（https://facebook.github.io/react/docs/more-about-refs.html）。

注意，无状态函数是没有实例化对象的，因此无法使用生命周期函数，也没有内部状态。所以如果你的组件需要使用生命周期函数或者内部状态，请使用类编写该组件。

组件、ReactElement 与组件实例的区别

因为组件、ReactElement 与组件实例这三个概念容易混淆，所以这里将它们放在一起对比介绍一下。

我们在 index.js 中将组件、ReactElement 与组件实例打印出来。

```
import React from 'react';
import { render } from 'react-dom';
import App from './src/App';

const componentInstance = render(<App />, document.querySelector('#root'));

...

console.log('组件、ReactElement与组件实例');
console.log(App);
console.log(<App />);
console.log(componentInstance);
```

运行程序后，可以看到在控制台中打印了一个名为 App 的类（JavaScript 中的类是基于 Function 原型生成的，因此这里打印了`function App()`）、一个`$$typeof`为`Symbol(react.element)`的对象和一个名为 App 的对象，这三个变量分别就是组件、ReactElement 和组件实例（见图 7-3）。

从图 7-3 中可以看出，组件、ReactElement 与组件实例是三个完全不同的概念。组件是一个函数或类，它决定了如何把数据变为视图；ReactElement 只是一个普通对象，它一个描述了组件实例或 DOM 节点；组件实例则是组件类的实例化对象。

```
组件、ReactElement与组件实例
function App() {
    _classCallCheck(this, App);

    return _possibleConstructorReturn(this, (App.__proto__ || Object.getPrototypeOf(App)).apply(this, arguments));
  }
▼ Object
    $$typeof: Symbol(react.element)
    _owner: null
    _self: null
    _source: null
  ▶ _store: Object
    key: null
  ▶ props: Object
    ref: null
  ▶ type: function App()
  ▶ __proto__: Object
▼ App
  ▶ _reactInternalInstance: ReactCompositeComponentWrapper
  ▶ context: Object
    getDOMNode: (...)
    isMounted: (...)
  ▶ props: Object
  ▶ refs: Object
    replaceProps: (...)
    replaceState: (...)
    setProps: (...)
    state: null
  ▶ updater: Object
  ▶ __proto__: App
```

图 7-3

组件实例的生灭：生命周期函数

生命周期函数是 React 组件类中一些函数钩子，它们在组件安装（也可以理解为渲染）、更新、卸载时被调用。那么生命周期函数和组件实例有什么关系呢？事实上，组件渲染、更新、卸载的过程正是组件实例初始化、更新、销毁的过程。

现将生命周期函数及其触发时刻介绍如下：

- componentWillMount在渲染前调用。
- componentDidMount在第一次渲染后调用。
- componentWillReceiveProps 在组件接收到一个新的 prop 时被调用。这个方法在第一次渲染时不会被调用。
- shouldComponentUpdate 返回一个布尔值。在组件接收到新的 props 或者 state 时被调用。在初始化时或者使用forceUpdate时不被调用。可以在你确认不需要更新组件时使用。
- componentWillUpdate在组件接收到新的 props 或者 state 但还没有 render 时被调用。在初始化时不会被调用。

- componentDidUpdate在组件完成更新后立即调用。在初始化时不会被调用。
- componentWillUnmount在组件从 DOM 中移除的时候立刻被调用。

这些方法的详细说明，可以参考官网文档（https://facebook.github.io/react/docs/component-specs.html#lifecycle-methods）。

下面一节将会通过一个例子来说明 React 组件方法中的this指向问题。同时请注意观察该例子中的生命周期函数的触发时刻。

React 组件中的 this

前面提到了某些组件方法中的this也指向组件实例。那么哪些组件方法中的 this 指向组件实例？React 组件中的 this 到底是什么？如何才能在自定义的组件方法中获取组件实例呢？下面将详细探讨这些问题。

React 组件的 this 到底是什么

编写下面这个组件，将其渲染出来，点击 h1 元素看看会发生什么。

```
import React from 'react';

const suffix = '被调用，this指向：';

export default class App extends React.Component {

  handler() {
    console.log(`handler${suffix}`, this);
  }

  render() {
    console.log(`render${suffix}`, this);

    return (
      <div>
        <h1 onClick={this.handler}>Hello world</h1>
      </div>
    );
  }
}
```



handler${suffix}中的反引号和${}是 ES2015 的模板字符串写法。模板字符串可以将上下文中的变量通过${}包裹写在字符串中，但是要注意字符串应该写在反引号里面。尽管这里可以使用+来拼接多个字符串，但在 Airbnb 编码规范中，推荐使用模板字符串替代使用+拼接字符串的写法。

结果是，render()函数中的this指向了组件实例，而handler()函数中的this则是一个null。为何会如此？React 组件中的 this 到底是什么？我们继续讲解。

JavaScript 函数中的 this

在讲解 React 组件中的this之前，我们先来说下 JavaScript 函数中的this。JavaScript 函数中的this不是在函数声明的时候而是在函数运行的时候定义的。这到底是什么意思呢？看代码：

```
var student = {
  speak: function() {
    console.log(this)
  }
};

student.speak()
var studentSpeak = student.speak
studentSpeak()
```

将这段代码放在 Chrome 的控制台里运行，可以看到student.speak()打印了student对象，因为this指向 student 对象。而studentSpeak()打印了window，因为this指向window。

这段代码非常形象地验证了，JavaScript 函数中的this不是在函数声明的时候而是在函数运行的时候定义的。

React 组件也遵循 JavaScript 的这种特性，所以组件方法的“调用者”不同会导致this不同。注意，这里的“调用者”是指函数执行时的当前对象。

“调用者”不同导致 this 不同

让我们分别在组件自带的生命周期函数以及自定义的handler()方法中打印this，并在render()方法里分别使用this.handler()、window.handler()、onClick={this.handler}这三种方法调用handler()，看看this会有什么不同。

src/App.js

```
/* eslint-disable no-useless-constructor */
import React from 'react';

const suffix = '被调用，this指向：';

export default class App extends React.Component {
  constructor(props) {
    super(props);
    // this.handler = this.handler.bind(this)
  }

  componentDidMount() {
    console.log(`componentDidMount${suffix}`, this);
  }

  componentWillReceiveProps() {
    console.log(`componentWillReceiveProps${suffix}`, this);
  }

  shouldComponentUpdate() {
    console.log(`shouldComponentUpdate${suffix}`, this);
    return true;
  }

  componentDidUpdate() {
    console.log(`componentDidUpdate${suffix}`, this);
  }

  componentWillUnmount() {
    console.log(`componentWillUnmount${suffix}`, this);
  }

  handler() {
    console.log(`handler${suffix}`, this);
  }
```

```
  render() {
    console.log(`render${suffix}`, this);

    this.handler();
    window.handler = this.handler;
    window.handler();

    return (
      <div>
        <h1 onClick={this.handler}>Hello world</h1>
        <p>不清楚组件、ReactElement、组件实例以及组件中的this是什么？打开控制台看
            看就明白了！ </p>
      </div>
    );
  }
}
```

index.js

```
import React from 'react';
import { render, unmountComponentAtNode } from 'react-dom';
import App from './src/App';

console.log('首次挂载');
let instance = render(<App />, document.getElementById('root'));

window.renderComponent = () => {
  console.log('挂载');
  instance = render(<App />, document.getElementById('root'));
};

window.setState = () => {
  console.log('更新');
  instance.setState({ foo: 'bar' });
};

window.unmountComponentAtNode = () => {
  console.log('卸载');
```

```
  unmountComponentAtNode(document.getElementById('root'));
};
```

上述代码做了以下几件事：

- 在生命周期函数中打印this。
- 使用this.handler()、window.handler()、onClick={this.handler}三种方法调用handler()，打印handler()中的this。
- 使用window.renderComponent、window.setState、window.unmountComponentAtNode三个方法封装组件的挂载、更新和卸载过程。

最后使用三个按钮触发组件的挂载、更新和卸载：

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <button onclick="window.renderComponent()">挂载</button>
    <button onclick="window.setState()">更新</button>
    <button onclick="window.unmountComponentAtNode()">卸载</button>
    <div id="root">
    </div>
    <script src="/static/bundle.js"></script>
  </body>
</html>
```

运行程序，然后依次单击“挂载”按钮、h1 元素（Hello world）、“更新”按钮、“卸载”按钮，结果发现：

- render()以及componentDidMount()、componentDidUpdate()等其他生命周期函数中的this都是组件实例。
- this.handler()的“调用者”为render()中的this，所以打印了组件实例。

- window.handler()的“调用者”为window，所以打印了window。
- onClick={this.handler}的“调用者”为事件绑定，来源多样，这里打印了null。

现在你应该明白，“组件方法的‘调用者’不同会导致this不同”这句话了。

面对这样混乱的场景，如果我们想在onClick中调用自定义的组件方法，并在该方法中获取组件实例，该怎么办呢？当然是转换上下文或者叫作绑定上下文了。

没有自动绑定

React.createClass 有一个内置的魔法，可以自动绑定所有的方法，使this指向组件的实例化对象。但是这对于 JavaScript 开发者来说有点奇怪，因为其他 JavaScript 类并没有这种特性。

所以 React 团队决定不在 React 组件类中实现自动绑定，把上下文转换的自由权交给开发者。我们通常在构造函数里绑定方法：

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.handler = this.handler.bind(this);
  }

  handler() {
    console.log("handler", this);
  }

  render() {
    console.log("render", this);
    return (
      <div>
        <h1 onClick={this.handler}>Hello world</h1>
      </div>
    );
  }
}
```

将this.handler()绑定上下文为组件实例后，this.handler()中的this将指向组件实例。此处也可以使用箭头函数或者在属性中临时绑定，不过 Airbnb 编码规范并不推荐在 JSX 的属性中使用bind临时绑定，所以尽量在构造函数中进行绑定或者使用箭头函数。

至此，React 的基础知识就介绍完了。当然，JSX 语法、state、props、context、ReactElement、组件实例这些知识都只是 React 知识体系中的一部分。在日常的开发过程中，如果遇到不能解决的问题，仍然需要查阅 React 官网文档。从下一章开始，我们将会介绍 Redux 的一些基础知识和注意事项。

总结

- ReactElement 是一个普通的对象，它描述了一个组件的实例或一个 DOM 节点及其属性。
- 组件实例是组件类的实例化对象，它通常被用来管理内部状态和处理生命周期函数。无状态函数是没有实例化对象的，因此无法使用生命周期函数，也没有内部状态。
- 生命周期函数是 React 组件类中一些函数钩子，它们在组件安装（也可以理解为渲染）、更新、卸载时被调用。
- React 组件生命周期函数中的`this`指向组件实例。自定义组件方法的`this` 会因“调用者”不同而不同。为了在组件的自定义方法中获取组件实例，需要手动绑定`this`到组件实例。

第8章

初识Redux

Action、reducer、state、store是Redux程序中的一些基本概念。本章通过一个完整的Redux小程序，介绍这些基本概念以及相关的注意事项。

例子

本章例子是一个简单的计数器程序，可以加减数值，并在数值更新时打印变化前后的state。

安装依赖并运行：

```
npm install
npm start
```

运行结果：

```
pre state: 0 next state: 1
pre state: 1 next state: 2
pre state: 2 next state: 1
```

Action

Action本质上是JavaScript普通对象。我们约定，action内使用一个字符串类型的type字段来表示将要执行的动作。多数情况下，type会被定义成字符串常量。另外，当应用规模越来越

大时，建议使用单独的模块或文件来存放 action。除了type字段外，action 对象的结构完全由你自己决定。

本章程序的触发是从发起 action 开始的：

src/app.js

```
store.dispatch({ type: 'INCREMENT' });
store.dispatch({ type: 'INCREMENT' });
store.dispatch({ type: 'DECREMENT' });
```

其中{ type: 'INCREMENT' }和{ type: 'DECREMENT' }就是 action。

Reducer

定义

Reducer 是个形式为(state, action) => state的纯函数，描述了 action 如何把 state 转变成下一个 state 。



“Reducer”这个名称来源于Array.prototype.reduce(reducer, ?initialValue)中的第一个参数reducer。reducer是一个累加器函数，它的参数是上个累加值和数组当前元素，然后通过计算返回本次的累加值。在 Redux 中，state 就是那个累加值，action 就是数组当前的元素。reduce以及map、filter等方法是函数式编程中十分常用的数组处理方法。在后续的章节中，我们会详细讲解这些数组处理方法。

```
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}
```

State 的形式取决于你，可以是基本类型、数组、对象，甚至是 Immutable.js 生成的数据结构。唯一的要点是当 state 变化时需要返回全新的对象，而不是修改传入的参数。

为何 reducer 要用纯函数？为何不能改变 state 参数，而要返回一个新的 state 对象？下面会一一解读这些问题。

纯函数

纯函数（Pure Function）是这样一种函数——输入/输出数据流全是显式（Explicit）的。显式的意思是，函数与外界交换数据只有一个唯一渠道——参数和返回值；函数从函数外部接受的所有输入信息都通过参数传递到该函数内部；函数输出到函数外部的所有信息都通过返回值传递到该函数外部。

纯函数不能访问外部变量，它能接触的“外地人”只有来自外部的参数。纯函数不能修改参数，因为这样做可能会把一些信息通过输入参数，夹带到外界。

Reducer 就是这样的函数，永远不要在 reducer 里做这些操作：

- 修改传入的参数。
- 执行有副作用（Side effects）的操作，如 API 请求和路由跳转。
- 调用非纯函数，如`Date.now()`或`Math.random()`。

函数副作用会给程序设计带来不必要的麻烦，产生难以查找的错误，并且降低程序的可读性。严格的函数式语言要求函数必须无副作用。

不能修改参数 state

让我们尝试做点违规的事情——修改 state 并返回。

```
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      state++
      return state;
    case 'DECREMENT':
      state--
      return state;
    default:
```

```
      return state;
  }
}
```

运行后发现，并没有出错，打印结果和之前一模一样。你会开始质疑这个规则。而其实这只是侥幸。

当 state 是对象时，修改参数 state 会影响程序的变更追踪。如果后面你使用了 react-redux 将 Redux 程序连接到 UI 组件上，你的组件将不能更新，因为 react-redux 无法察觉到任何 state 变化。

为何会发生这样的问题？因为在 JavaScript 中，对象是引用类型，当你改变了参数 state，变化前后的两个 state 将会指向一个地址，react-redux 就会以为这是两个相同的 state，因而不会执行渲染。

让我们将 state 改为对象类型，来看看这个违规行为会带来怎样的后果：

```
function counter(state = { val: 0 }, action) {
  switch (action.type) {
    case 'INCREMENT':
    state.val++;
      return state;
    case 'DECREMENT':
    state.val--;
      return state;
    default:
      return state;
  }
}
```

输入npm start，查看打印信息：

```
pre state: { val: 1 } next state: { val: 1 }
pre state: { val: 2 } next state: { val: 2 }
pre state: { val: 1 } next state: { val: 1 }
```

可以发现，变化前后的 state 是相同的。所以切记不要修改 state 参数。那么，如何返回一个新的对象？通常情况下，会使用Object.assign或者直接使用 Immutable.js 生成的不可变数据类型。这些内容后续会详细讲解。

Store

职能

Store 是个全局的对象，将 action 和 reducer 以及 state 联系在一起。Store 有以下职能：

- 维持应用的 state。
- 提供getState()方法获取 state。
- 提供dispatch(action)方法更新 state。
- 通过subscribe(listener)注册监听器。

创建

创建 store 需要从redux包中导入createStore这个方法：

```
import { createStore } from 'redux';
```

使用 reducer 纯函数作为第一个参数创建 store：

```
let store = createStore(counter);
```

本例中只有一个参数，但是可以将初始 state 作为第二个参数传入。修改 app.js 中的createStore函数，添加第二个参数 100 作为初始 state：

```
let store = createStore(counter, 100);
```

执行npm start，得到：

```
pre state: 100 next state: 101
pre state: 101 next state: 102
pre state: 102 next state: 101
```

获取与监听

创建完 store，使用它获取数据，并监听变化：

```
const store = createStore(counter);

let currentValue = store.getState();
```

```
store.subscribe(() => {
    const previousValue = currentValue;
    currentValue = store.getState();
    console.log('pre state:', previousValue, 'next state:', currentValue);
  }
);
```

以上代码做了这样几件事：

- 获取初始化的 state 到currentValue。
- 使用store.subscribe()方法监听变化。
- 在store.subscribe()的回调函数中，将currentValue传给previousValue作为先前的 state。
- 获取更新后的 state 到currentValue作为当前的 state。
- 打印变化前后的 state

如此一来，当发起 action 时，就会打印变化前后的 state：

```
pre state: 0 next state: 1
pre state: 1 next state: 2
pre state: 2 next state: 1
```

发起 action

Store 使用dispatch(action)方法发起 action，更新 state：

```
store.dispatch({ type: 'INCREMENT' });
store.dispatch({ type: 'INCREMENT' });
store.dispatch({ type: 'DECREMENT' });
```

当发起 action 后，就将 action 传进了 store 中，使用 reducer 纯函数执行更新。改变内部 state 唯一方法是dispatch一个 action。这样确保了视图和网络请求都不能直接修改 state，相反它们只能表达想要修改的意图，也就是dispatch 一个 action。

通向罗马的路有千万条，可惜最后都集中到dispatch这一条路上面了。集中处理的好处很明显，如果你在这条路上面放一个日志中间件，所有的变化都会被打印。而且不止打印日志这个功能，社区为我们提供了丰富的中间件，你可以选择需要的放在在dispatch这条路上。所有的变化，尽在你的掌握之中。

本章例子的完整代码如下：

src/app.js

```
import { createStore } from 'redux';

function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}

const store = createStore(counter);

let currentValue = store.getState();

const listener = () => {
  const previousValue = currentValue;
  currentValue = store.getState();
  console.log('pre state:', previousValue, 'next state:', currentValue);
};

store.subscribe(listener);


store.dispatch({ type: 'INCREMENT' });
store.dispatch({ type: 'INCREMENT' });
store.dispatch({ type: 'DECREMENT' });
```

总结

- Action 是个 JavaScript 对象，它是 store 数据的唯一来源。

- Reducer 是纯函数，不要在 reducer 中做这些事情：修改传入参数；执行有副作用的操作；调用非纯函数。
- Store 负责更新、查询、订阅 state 等多个工作。Store 是全局唯一的，它将 action、reducer、state 等联系在一起。

第9章

Action 创建函数与 Redux Thunk 中间件

Action 创建函数（action creator）就是创建 action 对象的函数，起初它只能返回 action 对象，但是通过一些中间件的加工后，action 创建函数将可以返回更多的类型。Action 创建函数和中间件的配合可以说是 Redux 中最有趣的部分了。本章将会介绍 action 创建函数的一些基本知识，还会使用 Redux Thunk 中间件去初步加工 action 创建函数。

例子

本章例子是基于第 8 章的示例程序改写的，除了普通的加减，还增加了奇数加一和延迟 1 秒加一的功能。

运行结果：

```
pre state: 0 next state: 1
pre state: 1 next state: 2
pre state: 2 next state: 1
pre state: 1 next state: 2
```

其中最后一行是延迟 1 秒后打印的。

Action 创建函数

编写

Action 创建函数，顾名思义就是创建 action 的函数，它是这样编写的：

```
function increment() {
  return { type: 'INCREMENT' };
}
function decrement() {
  return { type: 'DECREMENT' };
}
```

上述代码编写了两个 action 创建函数，分别返回加一和减一的 action 对象。

发起

如果要发起 action 创建函数，只需要将其返回结果传给dispatch()即可：

```
store.dispatch(increment());

store.dispatch(decrement());
```

如果不想每次都是用store.dispatch来发起 action 创建函数。可以将其封装在普通函数中，比如：

```
const boundIncrement = () => dispatch(increment());
const boundDecrement = () => dispatch(decrement())
```

如此一来，只需要调用boundIncrement()和boundDecrement()就可以发起 action 创建函数了。



在后面使用 react-redux 将 action 创建函数绑定到 React 组件上时，你可以使用redux提供的bindActionCreators()方法自动地将多个 action 创建函数封装成可以直接调用的普通函数。bindActionCreators()的详细用法，请参考官网 API 文档（http://redux.js.org/docs/api/bindActionCreators.html）。

意义

Action 创建函数的表面功能是创建 action 对象，但它的真正意义在于封装逻辑。比如说，你要实现一个奇数加一的功能，如果没有 action 创建函数，那么肯定是先判断当前数值是否是奇数，然后决定要不要发起加一的 action。但有了 action 创建函数后，就可以将这个逻辑封装在 action 创建函数中。然后程序要执行奇数加一时，只需发起这个 action 创建函数就行了。使用 action 创建函数封装逻辑是 React 与 Redux 配合的最佳实践。当所有的逻辑都被转移到 Redux 中后，React 就可以只负责渲染界面并发起 action 创建函数了。

Action 创建函数只能返回 action 对象，那么它是如何处理复杂的逻辑的呢？Redux 为了解决这个问题，引入了中间件的概念。中间件的功能之一就是可以让 action 创建函数返回更多的格式，以便于编写复杂的逻辑。下面会介绍如何使用 Redux Thunk 中间件增强 action 创建函数的功能。

Redux Thunk 中间件

功能

Redux Thunk 中间件可以让 action 创建函数先不返回 action 对象，而是返回一个函数。通过这个函数延迟dispatch或者只在满足指定条件的情况下dispatch。这个内部函数接受 store 的两个方法dispatch和getState作为参数。

```
function increment() {
  return { type: 'INCREMENT' };
}
function decrement() {
  return { type: 'DECREMENT' };
}
function incrementIfOdd() {
  return (dispatch, getState) => {
    const value = getState();
    if (value % 2 === 0) {
      return;
    }
```

```
    dispatch(increment());
  };
}
function incrementAsync(delay = 1000) {
  return dispatch => {
    setTimeout(() => {
      dispatch(increment());
    }, delay);
  };
}
```

上述代码中，incrementIfOdd返回的函数执行了条件判断。而incrementAsync返回的函数则延迟了dispatch。

安装激活

如果你需要在 action 创建函数中返回函数，执行一些条件判断或者延迟操作，那么首先要安装 Redux Thunk 中间件：

```
npm install --save redux-thunk
```

然后引入thunk和applyMiddleware，并在createStore中激活：

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';

const store = createStore(counter, applyMiddleware(thunk));
```

完整的代码如下：

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';

function increment() {
  return { type: 'INCREMENT' };
}
function decrement() {
  return { type: 'DECREMENT' };
}
function incrementIfOdd() {
```

```
  return (dispatch, getState) => {
    const value = getState();
    if (value % 2 === 0) {
      return;
    }

    dispatch(increment());
  };
}
function incrementAsync(delay = 1000) {
  return dispatch => {
    setTimeout(() => {
      dispatch(increment());
    }, delay);
  };
}

function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}


const store = createStore(counter, applyMiddleware(thunk));

let currentValue = store.getState();
store.subscribe(() => {
  const previousValue = currentValue;
  currentValue = store.getState();
  console.log('pre state:', previousValue, 'next state:', currentValue);
}
);
```

```
store.dispatch(increment());

store.dispatch(incrementIfOdd());

store.dispatch(incrementAsync());

store.dispatch(decrement());
```

上述代码做了这样几件事：

- 编写了四个 action 创建函数，即`increment`、`decrement`、`incrementIfOdd`、`incrementAsync`，分别用于加一、减一、奇数加一和异步加一。
- 编写了一个名为`counter`的 reducer 纯函数，用于更新 state。
- 使用`createStore`创建 store，并激活 Redux Thunk 中间件。
- 使用`store.subscribe()`监听变化，打印更新前后的 state。
- 使用`store.dispatch()`分别发起四个 action 创建函数。

中间件的使用拓展了 Redux 的功能，本来 action 创建函数只能返回一个 action 对象，因为有了 Thunk 中间件，现在可以返回一个函数了。事实上，你可以通过自己编写 Redux 中间件任意地定义 action 创建函数的返回格式，这些内容我们后续会详细讲解。

总结

- Action 创建函数就是创建 action 的函数。
- 如果要发起 action 创建函数，只需要将其返回结果传给`dispatch()`。
- Redux Thunk 中间件可以让 action 创建函数先不返回 action 对象，而是返回一个函数。通过这个函数延迟`dispatch`或者只在满足指定条件的情况下`dispatch`。
- 激活 Redux Thunk 中间件，只需要在`createStore`中加入`applyMiddleware(thunk)`即可。

第二部分

进阶篇

第10章

React 与 Redux 的连接：手动连接

本章将以计数器程序为例，介绍如何手动将 Redux 连接到 React 组件中。本章没有引入 action 创建函数，也没有做渲染方面的性能优化，仅仅只是为了让读者体会 React 与 Redux 的连接过程。在实际开发中，应该使用 react-redux 进行 React 与 Redux 的连接。

例子

本章例子是 Redux 官方的计数器程序。前两个按钮完成加减功能；第三个按钮，如果当前数字是奇数则加一；第四个按钮是异步加一（延迟 1 秒），如图 10-1 所示。

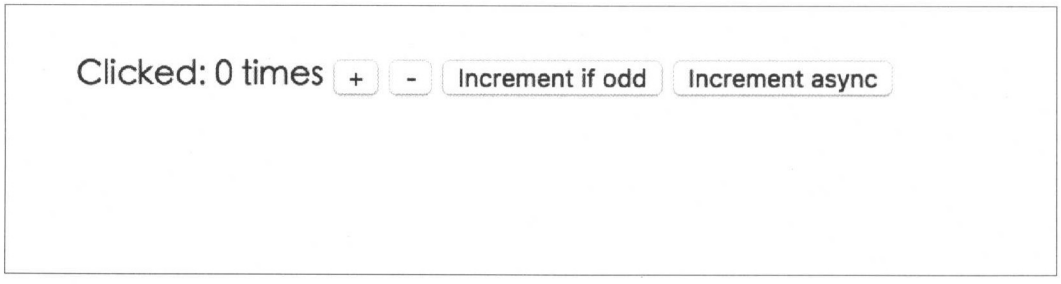


图 10-1

手动连接的步骤

使用 React 与 Redux 编写一个计数器程序的思路如下所述。

- 编写一个 React 组件来渲染界面，包括四个按钮和一个数值。
- 使用 Redux 来管理状态，即将计数器的当前数值托管在 Redux 的 state 中，还要提供加一和减一两个 action 来更新这个 state。
- 将 Redux 的 state 和发起 action 的方法通过 props 传递到组件中，并将 state 渲染为数值，将两个方法注册到按钮的点击事件上。
- 最后渲染组件，并在每次 state 被更新时重新渲染组件。

下面几节将会介绍上述步骤的具体细节。

编写 React 组件

编写 React 组件是实现计数器程序的第一步。事实上，后面所有程序开发的第一步都是编写 React 组件。编写计数器组件的步骤如下：

1. 设计组件的 props 输入为`value`、`onIncrement`、`onDecrement`，分别代表当前数值、加一方法、减一方法，并限制它们的类型。

 components/Counter.js

   ```
   ...

   Counter.propTypes = {
     // value必须为数字，且必须存在
     value: PropTypes.number.isRequired,
     // onIncrement必须为fucntion，且必须存在
     onIncrement: PropTypes.func.isRequired,
     onDecrement: PropTypes.func.isRequired
   };
   ...
   ```

2. 在`render`方法中获取 props 输入值，然后渲染界面，包括一个数值和四个按钮。

components/Counter.js

```
import React, { Component, PropTypes } from 'react';

class Counter extends Component {
  ...
  render() {
    const { value, onIncrement, onDecrement } = this.props;
    return (
      <p>
        Clicked: {value} times
        {' '}
        <button onClick={onIncrement}>+</button>
        {' '}
        <button onClick={onDecrement}>-</button>
        {' '}
        <button onClick={this.incrementIfOdd}>Increment if odd</button>
        {' '}
        <button onClick={this.incrementAsync}>Increment async</button>
      </p>
    );
  }
}
```

3. 编写奇数加一和异步加一的组件方法，并将方法中的 this 绑定到组件实例上。

components/Counter.js

```
import React, { Component, PropTypes } from 'react';

class Counter extends Component {
  constructor(props) {
    // 子类在获取this前必须调用super
    super(props);
    this.incrementAsync = this.incrementAsync.bind(this);
    this.incrementIfOdd = this.incrementIfOdd.bind(this);
  }
```

```
  incrementIfOdd() {
    if (this.props.value % 2 !== 0) {
      this.props.onIncrement();
    }
  }

  incrementAsync() {
    setTimeout(this.props.onIncrement, 1000);
  }

  ...
}

...
```

至此，React 组件就编写完成了。

编写 Redux

因为本例没有引入 action 创建函数，所以编写 Redux 只需要两步：编写 reducer 和创建 store。

1. 编写 reducer，使其可以接受加一/减一的 action，并据此更新 state。

reducers/index.js

```
export default function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}
```

2. 创建 store。

index.js

```
...
import counter from './reducers';

const store = createStore(counter);
...
```

将 Redux 手动连接到 React 组件

将 Redux 手动连接到 React 组件也需要两步：首先将 state 和发起 action 的方法连接到组件中，然后渲染组件并监听变化。后面只要点击按钮就会发起 action，再次渲染。

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { createStore } from 'redux';
import Counter from './components/Counter';
import counter from './reducers';

const store = createStore(counter);
const rootEl = document.getElementById('root');

function render() {
  ReactDOM.render(
    <Counter
      value={store.getState()}
      onIncrement={() => store.dispatch({ type: 'INCREMENT' })}
      onDecrement={() => store.dispatch({ type: 'DECREMENT' })}
    />,
    rootEl
  );
}

render();
store.subscribe(render);
```

到此为止，这个程序就编写完成了。在命令行中启动服务，打开浏览器看看你的第一个 React 与 Redux 程序吧！

手动连接的缺点

因为本章的程序比较简单，所以可以手动连接 React 和 Redux。但是在真实的项目中，手动连接并不可取，它有以下两个明显的缺点：

- 无法直接给里面的组件传递 state 和方法。
- 任意 state 的变化都会导致整个组件树的重新渲染，没有优化性能。

尽管我们可以通过自己编写代码来避免上述缺点，但是使用第三方库或许是更好的选择。react-redux 就是专门为此设计的，它不仅可以给组件树中任一组件绑定 state 和方法，还进行了性能优化，可以避免不必要的重新渲染。下一章我们将学习使用 react-redux 将 Redux 优雅地连接到 React 组件上。

总结

- 对于简单的程序而言，可以手动连接 React 和 Redux。但在真实项目中，手动连接并不可取，它有两个明显缺点：无法直接向里面的组件传递 state 和方法；任意 state 的变化都会导致整个组件树的重新渲染，没有优化性能。
- react-redux 不仅可以给组件树中任一组件绑定 state 和方法，还进行了性能优化，可以避免不必要的重新渲染。在实际开发中，应该使用 react-redux 进行 React 与 Redux 的连接。

第11章

React 与 Redux 的连接：使用 react-redux 连接

本章将介绍如何使用 react-redux 进行 React 与 Redux 的连接。与第 10 章的手动连接相比，使用 react-redux 进行连接不仅更加方便，而且这个工具还内置了性能优化的功能，帮助程序减少不必要的性能消耗。

例子

本章例子仍然是计数器程序，不过使用 react-redux 进行连接，并且演示了多种写法（见图 11-1）。另外，本例还使用了 action 创建函数来封装奇数加一和异步加一的逻辑。

连接步骤

在进行连接前，需要下载安装 react-redux。

```
npm install --save react-redux
```

安装完成后，只需两步即可将 Redux 连接到 React 中：

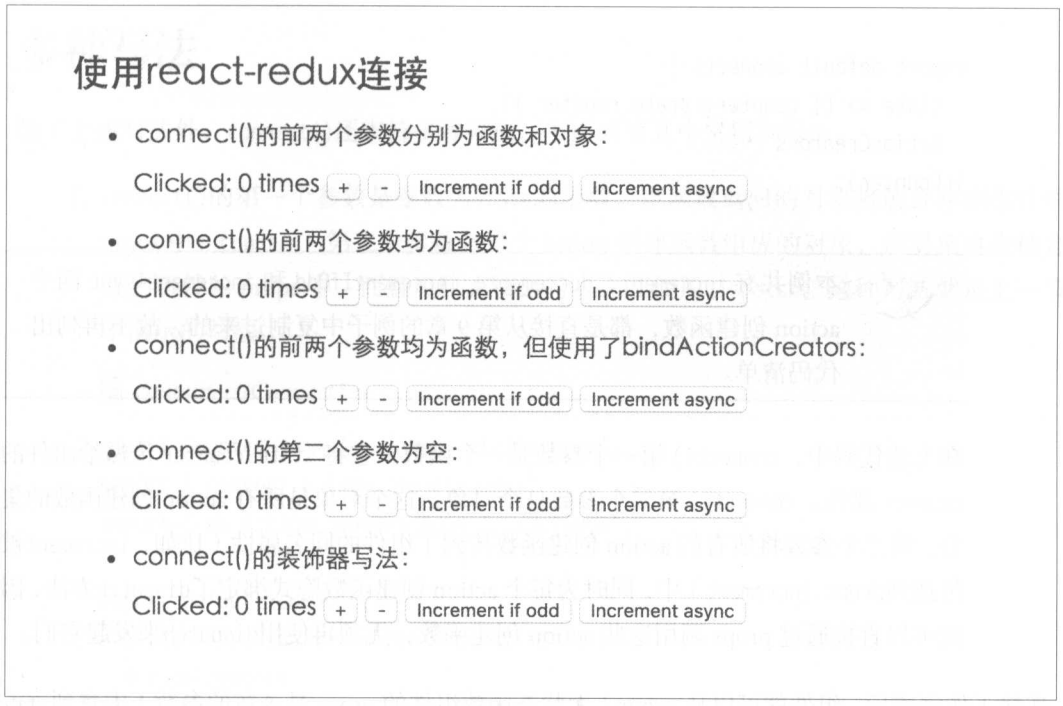


图 11-1

1. 在所有组件的顶层使用Provider组件给整个程序提供store。

index.js

```
import { Provider } from 'react-redux';
...

ReactDOM.render(
  <Provider store={store}>
    ...
  </Provider>, rootEl);
```

2. 使用connect()将 state 和 action 创建函数绑定到组件中。

containers/Connect1.js

```
import Counter from '../components/Counter';
import { connect } from 'react-redux';
import * as ActionCreators from '../actions';
```

```
export default connect(
  state => ({ counter: state.counter }),
  ActionCreators
)(Counter);
```



本例共有 increment、decrement、incrementIfOdd 和 incrementAsync 四个 action 创建函数，都是直接从第 9 章的例子中复制过来的，故不再列出代码清单。

在上述代码中，connect() 第一个参数是一个函数，它将 state.counter 传递给组件的 counter 属性。connect() 第二个参数是个对象，这个对象是所有 action 创建函数的集合。第二个参数将所有的 action 创建函数传到了组件的同名属性（比如，increment被传递到props.increment）中，同时为每个 action 创建函数隐式绑定了dispatch方法，因此可以直接通过 props 调用这些 action 创建函数，无须再使用dispatch来发起它们。

连接工作完成后，组件就可以从 props（无状态函数组件的 props 是函数的参数）中拿到 state 和 action 创建函数了，而且一旦 state 发生变化，被连接的组件也会自动重新渲染。

components/Counter.js

```
function Counter({ counter, increment, decrement, incrementIfOdd, incrementAsync }) {
  return (
    <p>
      Clicked: {counter} times
      {' '}
      <button onClick={increment}>+</button>
      {' '}
      <button onClick={decrement}>-</button>
      {' '}
      <button onClick={incrementIfOdd}>Increment if odd</button>
      {' '}
      <button onClick={() => incrementAsync()}>Increment async</button>
    </p>
  );
}
```

多种写法

除了上述写法外，connect()还支持多种写法，下面列举几个常用的写法。

1. connect()的第一个参数是参数为state的函数，该函数返回的对象将被合并到组件的 props 中。connect()第二个参数是多个 action 创建函数组成的对象，该对象也会被合并到组件的 props 中，而且可以直接运行，无须使用dispatch。这种写法就是上一节中演示的写法。

containers/Connect1.js

```
import Counter from '../components/Counter';
import { connect } from 'react-redux';
import * as ActionCreators from '../actions';

export default connect(
  state => ({ counter: state.counter }),
  ActionCreators
)(Counter);
```

2. connect()的第一个参数不变。connect()第二个参数是参数为dispatch的函数，该函数返回的对象也会被合并到组件的 props 中。注意，该写法不会自动为 action 创建函数绑定dispatch方法，所以需要我们手动来绑定。

containers/Connect2.js

```
import Counter from '../components/Counter';
import { connect } from 'react-redux';
import { increment, decrement, incrementIfOdd, incrementAsync } from '../actions
    ';

export default connect(
  state => ({ counter: state.counter }),
  dispatch => ({
    increment: () => dispatch(increment()),
    decrement: () => dispatch(decrement()),
    incrementIfOdd: () => dispatch(incrementIfOdd()),
    incrementAsync: () => dispatch(incrementAsync()),
  })
```

```
)(Counter);
```

3. connect()的第一个参数不变。connect()第二个参数是参数为dispatch的函数，但是函数的返回值使用了redux的bindActionCreators()来减少样板代码。

containers/Connect3.js

```
import Counter from '../components/Counter';
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';
import * as ActionCreators from '../actions';

export default connect(
  state => ({ counter: state.counter }),
  dispatch => bindActionCreators(ActionCreators, dispatch)
)(Counter);
```

4. connect()的第一个参数不变。connect()第二个参数为空。此时connect()将会自动给组件传递一个dispatch，让组件自己使用dispatch来发起 action 创建函数。

containers/Connect4.js

```
import React, { PropTypes } from 'react';
import { connect } from 'react-redux';
import { increment, decrement, incrementIfOdd, incrementAsync } from '../actions
    ';

function Counter({ counter, dispatch }) {
  return (
    <p>
      Clicked: {counter} times
      {' '}
      <button onClick={() => dispatch(increment())}>+</button>
      {' '}
      <button onClick={() => dispatch(decrement())}>-</button>
      {' '}
      <button onClick={() => dispatch(incrementIfOdd())}>Increment if odd</button
          >
      {' '}
      <button onClick={() => dispatch(incrementAsync())}>Increment async</button>
```

```
      </p>
    );
  }

  Counter.propTypes = {
    counter: PropTypes.number.isRequired,
    dispatch: PropTypes.func.isRequired
  };

  export default connect(
    state => ({ counter: state.counter })
  )(Counter);
```

5. 使用装饰器写法将connect()写在组件类声明上面。装饰器写法和前面的写法只是语法上的区别，其他并无二致。但是，如果使用装饰器写法，就不能使用无状态函数来编写组件了，因为装饰器需要添加在类声明上。另外，还必须使用static propTypes = {...}这种静态属性的写法来声明 props 类型。

containers/Connect5.js

```
import React, { Component, PropTypes } from 'react';
import { connect } from 'react-redux';
import * as ActionCreators from '../actions';

// 装饰器应该写在类声明上面
@connect(
  state => ({ counter: state.counter }),
  ActionCreators
)
class Counter extends Component { // eslint-disable-line

  static propTypes = {
    counter: PropTypes.number.isRequired,
    increment: PropTypes.func.isRequired,
    incrementIfOdd: PropTypes.func.isRequired,
    incrementAsync: PropTypes.func.isRequired,
    decrement: PropTypes.func.isRequired
  };
```

```
  render() {
    const { counter, increment, decrement, incrementIfOdd, incrementAsync } =
        this.props;
    return (
      <p>
        Clicked: {counter} times
        {' '}
        <button onClick={increment}>+</button>
        {' '}
        <button onClick={decrement}>-</button>
        {' '}
        <button onClick={incrementIfOdd}>Increment if odd</button>
        {' '}
        {/* 这里必须写成箭头函数，否则incrementAsync中的delay参数将会是
            SyntheticEvent的实例*/}
        <button onClick={() => incrementAsync()}>Increment async</button>
      </p>
    );
  }
}

export default Counter;
```



因为 ES2015 是不支持static和@装饰器这两种写法的，所以你还需要安装stage-0预设和transform-decorators-legacy插件。stage-0预设包含一系列插件可以让你用上试验阶段的语言特性。babel-plugin-transform-decorators-legacy则是给装饰器用的插件。

无论是否使用装饰器，connect都是一定要用的。关于connect方法的详细用法，React Redux 的官网文档（https://github.com/reactjs/react-redux）是最好的参考，本章以连接原理和注意事项为重点，对 API 文档不再赘述。

连接原理

仅仅会用 react-redux 是不够的，如果想在使用时做到心里有数，遇到异常时不会惊慌失措，那么就必须了解 react-redux 的内部连接原理。下面几节将分别介绍Provider和connect的工作原理。

Provider 工作原理

Provider只是一个 React 组件，它的职能是通过 context 将 store 传递给子组件。因为Provider组件是通过 context 传递 store 的，所以里面的组件，不管跨多少级，都可以通过connect()方法获取 store 并进行连接。下面是它的源代码：

node_modules/react-redux/src/components/Provider.js（部分代码）

```
export default class Provider extends Component {
  getChildContext() {
    return { store: this.store }
  }
  ...
}

...

Provider.propTypes = {
  store: storeShape.isRequired,
  ...
}
Provider.childContextTypes = {
  store: storeShape.isRequired
}
```

可以看到Provider只是一个使用 context 传递数据的 React 组件而已。如果你忘记了 context 的相关知识，可以回顾前面章节的内容来复习。

connect 工作原理

connect是一个嵌套函数。运行connect()后，会生成一个高阶组件（Higher-order Components）。该高阶组件接受一个组件作为参数再次运行，会生成一个新组件。



高阶组件（Higher-order Components）是一个函数。它接受 React 组件作为参数，并返回一个新的 React 组件。高阶组件的意义在于可以让多个组件共同使用一些逻辑。比如，react-redux 将 Redux 与 React 组件的连接逻辑封装在高阶组件connect()中，然后就可以方便地给每个需要连接的组件使用了。

高阶组件新生成的组件叫Connect组件。Connect组件从 context 中获取了来自Provider的 store，然后从 store 中获取 state 和 dispatch，最后将 state 和经过 dispatch 加工的 action 创建函数连接到组件上，并在 state 变化时重新渲染组件。注意，假如一个页面中有多个被connect()连接的组件，这些组件只会在自己对应的 state 变化时重新渲染，不会相互干扰。因为 react-redux 在重新渲染组件前会检测传入组件的数据是否变化，如果没有变化，就不会执行渲染。如此一来，我们就不用担心页面中某一个组件数据的变化，会“连累”其他组件重新渲染了，只需要将不同的数据用不同的connect()隔离开即可。

react-redux 的连接原理就是这样。在开发中遇到问题，除了提交 Issue 之外，更应该多了解源码，这样才能做到融会贯通、熟练掌握。

展示组件与容器组件

如果你的页面中包含多个组件，是否应该将每个组件都连接到 Redux 中呢？当然不是。Redux 通常只与靠近顶层的组件进行连接，这些组件再将 state 和 action 创建函数通过 props 传给里面的组件。为了区分这两种组件，我们将与 Redux 连接的组件称为容器组件（container components），将里面的组件称为展示组件（presentational components）。这两类组件的区别如表 11-1 所示。

本例中的Connect1、Connect2、Connect3、Connect3和Connect4被放在 containers 目录中作为容器组件，而Counter组件则被放在 components 目录中作为展示组件。将组件分为展示组件和容器组件会带来很多便利，比如，可以更好地分离关注点、使组件拥有更高的可复用性等。

表 11-1

	展示组件	容器组件
功能	决定程序如何显示（模板、样式）	决定程序如何运作（数据获取、状态更新）
是否连接 Redux	否	是
读取数据	从 props 中读取数据	订阅 Redux 的 state
更新数据	调用来自 props 的回调函数	发起 Redux 的 action
生成方式	手动编写	大多通过 React Redux 自动生成

虽然容器组件与展示组件的约定是 React 与 Redux 开发中的最佳实践，但是这个约定并非需要死板地遵循。事实上，containers 目录中也可以放置没有连接 Redux 的组件，因为有些页面就是不需要连接 Redux 的，但它仍然是其他组件的“容器”。而 components 目录中也可以放置连接 Redux 的组件，比如后面会讲到的 ReduxForm 生成的表单组件。但在绝大多数情况下，我们都应该将 Redux 连接在组件顶层，不要让里面的组件感受到 Redux 的存在。

如果现在你还对使用 react-redux 有疑问，也没有关系，毕竟计数器的例子还是太过简单。在后面的章节中，我们会在更复杂的程序中使用 react-redux，相信看过几个例子的演示后，你一定可以熟练使用它。

总结

- `Provider`组件负责给程序提供 store，而`connect()`则负责生成新的名为`Connect`的组件，`Connect`组件在 context 中拿到 store 后，从 store 中获取 state 和 dispatch，最后将 state 和经过 dispatch 加工的 action 创建函数连接到组件上。
- 绝大多数情况下，我们都应该将 Redux 连接在组件顶层，不要让里面的组件感受到 Redux 的存在。

第12章

实现撤销/重做

本章将会介绍如何在 Redux 应用中快速实现撤销/重做，同时还会讲解 Redux 开发者工具的相关知识。

例子

本章例子仍然是计数器程序，不过增加了撤销/重做功能。另外，还会使用 Redux 开发者工具来追踪、回放程序状态（见图 12-1）。

撤销/重做

实现撤销/重做的功能非常简单，只需要使用`redux-undo`提供的`undoable`和`includeAction`配置 reducer 就可以了。

reducers/index.js

```
import { combineReducers } from 'redux';
import undoable, { includeAction } from 'redux-undo';
import counter from './counter';
import { INCREMENT_COUNTER, DECREMENT_COUNTER, UNDO_COUNTER, REDO_COUNTER } from '../
    actions';
```

```
const rootReducer = combineReducers({
  counter: undoable(counter, {
    filter: includeAction([INCREMENT_COUNTER, DECREMENT_COUNTER]),
    limit: 10,
    debug: true,
    undoType: UNDO_COUNTER,
    redoType: REDO_COUNTER
  })
});

export default rootReducer;
```

undoable 是个高阶函数（Higher-Order Function），这意味着可以使用函数作为它的参数，然后生成一个新的函数。



图 12-1



高阶函数是函数式编程中的一个概念，它可以接收其他函数作为参数，然后返回一个新的函数。高阶函数以及高阶组件都是为了增强函数或组件的功能而设计的。一般情况下，生成的新函数或组件不会失去原有的功能。

本来这里应该使用counter 这个纯函数作为 reducer，不过我们利用高阶函数，将counter 作为参数传入，通过配置创建了一个更高级的 reducer。下面将介绍配置选项的具体含义：

- filter 用于指定需要过滤的 action。这里只撤销和重做加/减 action，也就是INCREMENT 和DECREMENT 。
- limit是次数限制，当撤销或者重做的次数超过这个指定值，将无法继续撤销或者重做。
- debug 用于设定是否调试代码，激活状态下会打印相关的信息。
- undotype和redotype是撤销和重做的 action。如此一来，只需要触发撤销/重做的 action 便可以实现撤销/重做功能。

Action 创建函数和组件的代码如下：

actions/index.js

```
export const INCREMENT_COUNTER = 'INCREMENT_COUNTER';
export const DECREMENT_COUNTER = 'DECREMENT_COUNTER';
export const UNDO_COUNTER = 'UNDO_COUNTER';
export const REDO_COUNTER = 'REDO_COUNTER';

export function increment() {
  return {
    type: INCREMENT_COUNTER,
  };
}

export function decrement() {
  return {
    type: DECREMENT_COUNTER,
  };
}

export function undo() {
```

```
  return {
    type: UNDO_COUNTER
  };
}

export function redo() {
  return {
    type: REDO_COUNTER
  };
}
```

containers/Counter.js

```
import React, { PropTypes } from 'react';
import { connect } from 'react-redux';
import * as ActionCreators from '../actions';

function Counter({ increment, decrement, undo, redo, value }) {
  return (
    <p>
      Clicked: {value} times
      {' '}
      <button onClick={increment}>+</button>
      {' '}
      <button onClick={decrement}>-</button>
      {' '}
      <button onClick={undo}>Undo</button>
      {' '}
      <button onClick={redo}>Redo</button>
    </p>
  );
}

Counter.propTypes = {
  increment: PropTypes.func.isRequired,
  decrement: PropTypes.func.isRequired,
  undo: PropTypes.func.isRequired,
  redo: PropTypes.func.isRequired,
```

```
  value: PropTypes.number.isRequired
};

export default connect(
  state => ({ value: state.counter.present }),
  ActionCreators
)(Counter);
```

注意，由于高阶函数undoable的作用，此时的state.counter已经不是计数器的当前数值了，应该将其改为state.counter.present。

这个容器组件在之前的基础上添加了两个按钮，分别用于触发撤销和重做的 action。单击撤销/重做的按钮，看看会发生什么。如果开启了debug，可以打开控制台查看打印信息。

至此，撤销/重做的功能就实现了。

Redux 开发者工具

在前面的章节中，我们学习使用了 React 开发者工具。无独有偶，Redux 也有自己的开发者工具，我们来学习安装和使用它。

安装

安装 Redux 开发者工具有多种方式，可以像普通的 React 组件一样，将其渲染出来，也可以使用 Chrome 拓展。为了方便，我们选择后一种方法。

首先，在 Chrome 的商店里面搜索“Redux DevTools”找到它，并在将其添加到浏览器中，然后在创建 store 时激活：

index.js

```
const store = createStore(counter, compose(
      applyMiddleware(thunk),
      window.devToolsExtension ? window.devToolsExtension() : f => f
    ));
```

compose() 是 Redux 提供的一个 API，可以帮助我们将丑陋的嵌套函数写成平行的样子。



compose()使用了函数合成（Function Composition）的思想。函数合成是函数式编程中的概念，它接受多个函数作为参数并返回一个新函数。新函数按照传入的参数顺序，从右往左依次执行，前一个函数的返回值是后一个函数的输入值。下面是函数合成的一个简单示例：

```
const absAndRound = compose(Math.round, Math.abs)
// compose写法等同于下面的函数嵌套
const absAndRound = val => Math.round(Math.abs(val))
```

applyMiddleware(thunk) 用于激活中间件，window.devToolsExtension ? window.devToolsExtension() : f => f 用于激活 Redux 开发者工具。

接下来，打开 Chrome 开发者工具，单击“redux”选项卡就可以看见 Redux 开发者工具的界面了。

使用

Redux 开发者工具拥有丰富的功能，它不仅提供了日志监控（Log monitor）、检测器（Inspector）和图表（Chart）三个界面来帮助开发者实时追踪程序中 action 和 state 的每一个变化，还在底部工具栏里提供了 action 发起器（Dispatcher）、时间滑块（Slider）、导入/导出（Import/Export）、远程（Remote）等多个按钮来实现手动发起 action、时间旅行调试、导入/导出程序状态、远程调试等炫酷的功能。下面将会介绍上述界面和功能的具体细节。

日志监控界面

首先，看看日志监控界面，将工具切换到 Log monitor。在浏览器中单击计数器程序的按钮，可以发现计数器程序中发起的 action 被打印了出来。这些 action 信息以日志列表的形式出现在日志监控界面中，列表中每项的标题是 action 名称（即 action 对象的类型），每项的内容是对应的 state tree（见图 12-2）。



因为 Redux 中的 state 多为树状的对象，因此被称为“state tree”。

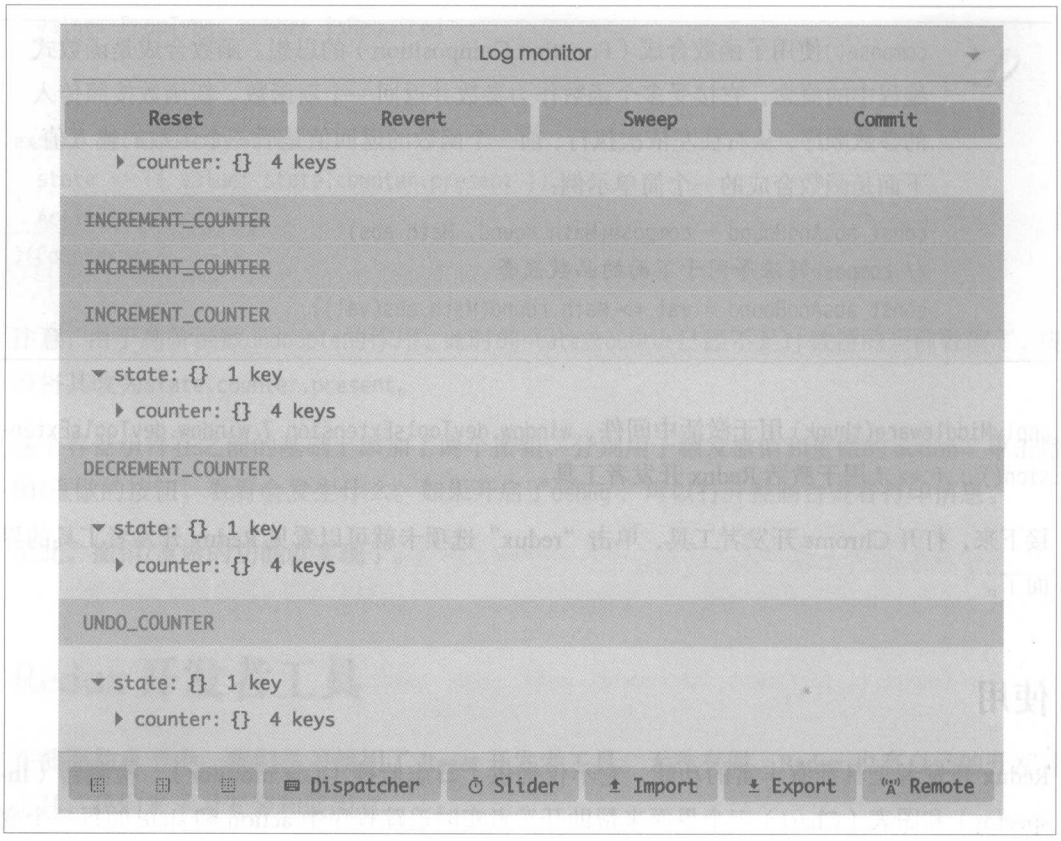


图 12-2

同时这个界面上还有四个按钮：

- **Sweep** 可以移除日志中所有失效的 action。通过单击日志列表中的 action 名称，可以将该 action 变为失效状态。此时单击“Sweep”按钮就可以将这个失效的 action 从面板中移除。
- **Commit** 类似于 Git 中的commit 命令，它可以从日志列表中移除所有的 action，并将当前的 state 作为初始化 state。
- **Revert** 可以移除所有的 action，将 state 恢复到上次提交（Commit）时的 state。
- **Reset** 可以移除所有的 action 并让你的 store 恢复到初始化状态，包括已经被提交的 action 也会被重置。

如果上面的图文不足以让你理解日志监控界面的功能用法，强烈建议运行本章例子，然后打开 Redux 开发者工具亲自体验一番。

检测器界面

切换到检测器界面（Inspector），可以看到共有左右两个面板。左侧面板是被发起的 action 列表，单击左侧列表中某个 action，会在右侧显示该 action 的详情。右侧面板中包含四个选项卡：Action、Diff、State 和 Test，分别用于显示 action 对象的详细内容、本次 action 引起的 state 变化、本次 action 发起后的最终 state 以及生成的测试代码（见图 12-3）。

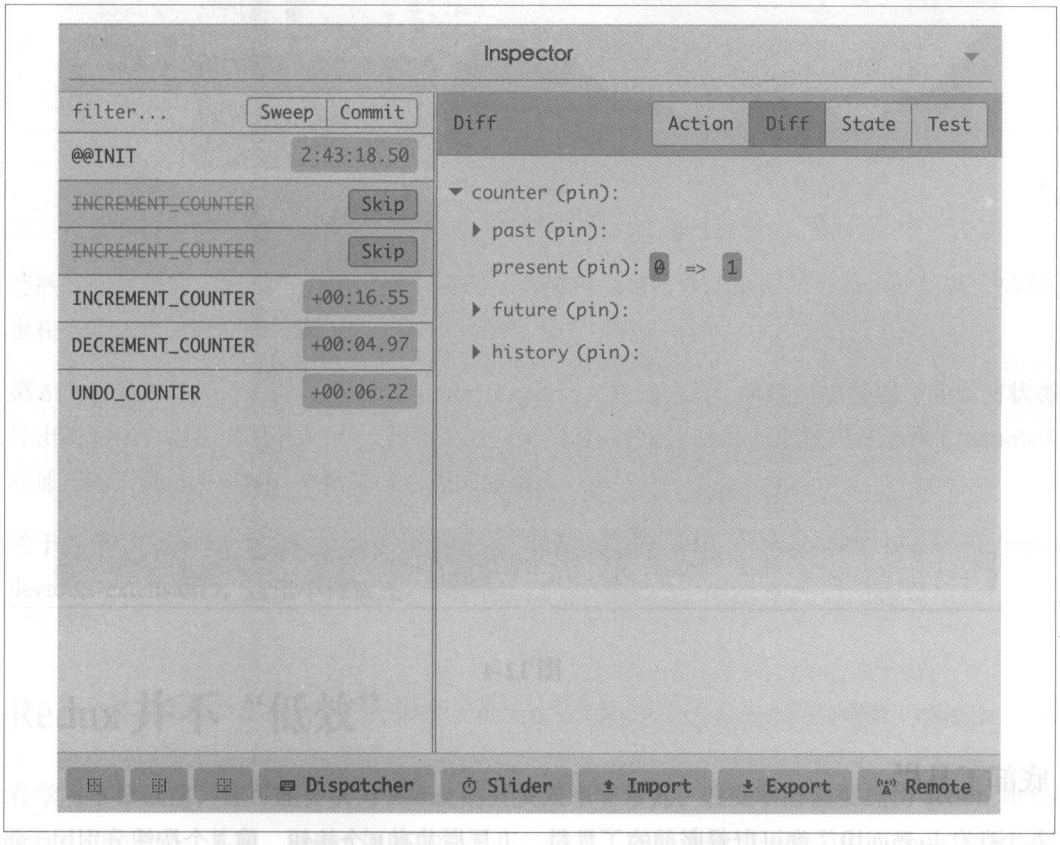


图 12-3

这里只给出 Diff 选项卡的内容截图，其他选项卡下的内容请自行查看。

图表界面

切换到图表界面（Chart），可以看到所有变化记录都被可视化地显示了，当鼠标滑过时还会弹出详细信息，如图 12-4 所示。

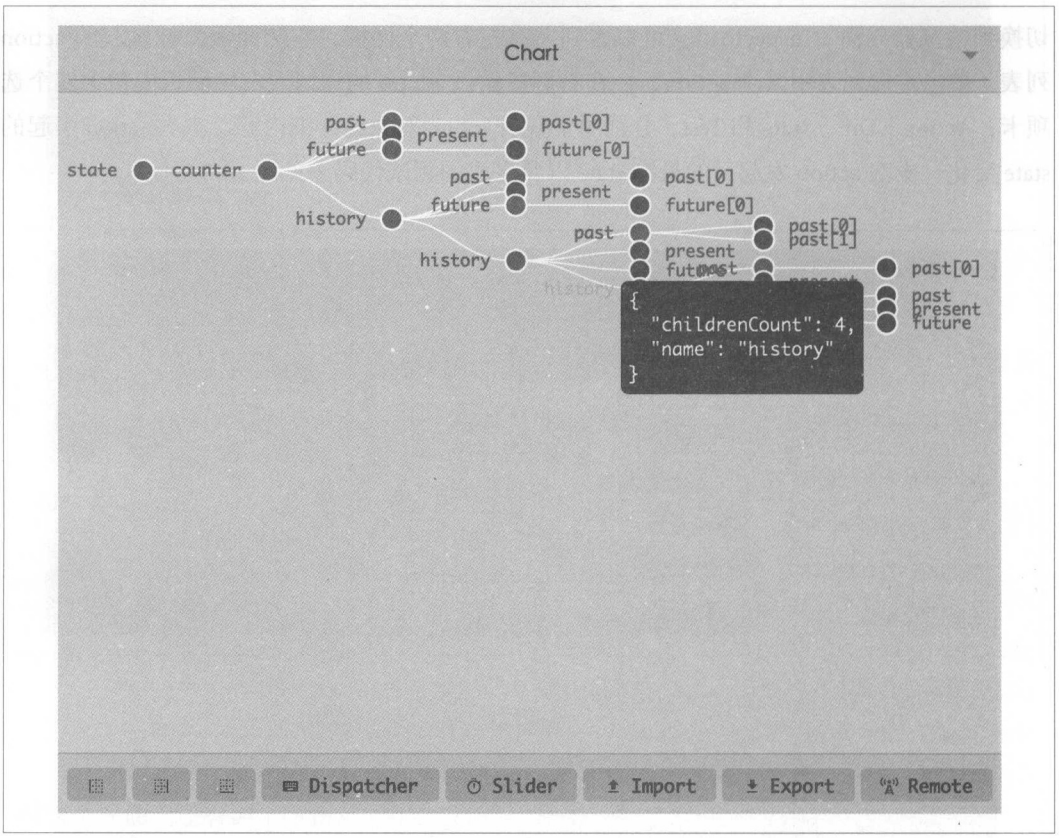


图 12-4

底部工具栏

在上述任一界面中，都可以看底部的工具栏。工具栏共有 8 个按钮，前 3 个按钮分别用于将 Redux 开发者工具以独立窗口的形式移动到桌面左侧、右侧和底部。第 4 个按钮是 action 发起器（Dispatcher），使用它可以给程序手动地发起 action（见图 12-5）。

第 5 个按钮是时间滑块（Slider），使用它可以实现时间旅行。时间旅行的意义在于方便调试，比如可以回退到 Bug 产生前的那一步，然后重现错误（见图 12-6）。

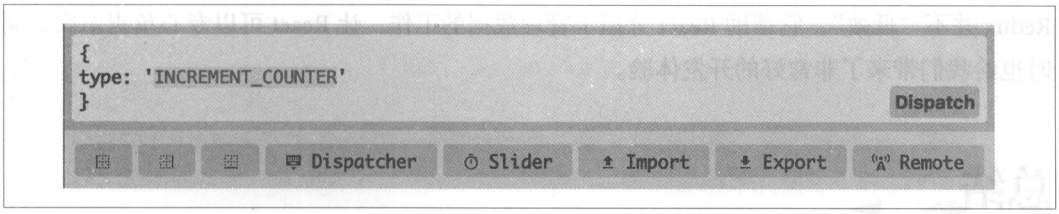


图 12-5

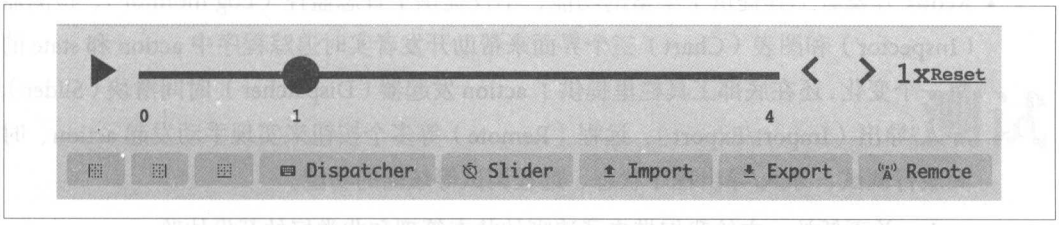


图 12-6

当然时间旅行并不只是可以通过滑块实现，当你在检测器界面单击某个 action 使其失效时，也相当于进行了时间旅行调试。

第 6 个和第 7 个按钮是导入/导出（Import/Export），通过这两个按钮可以将程序的当前状态导出为 JSON 文件，在需要时导入程序中以恢复当时的状态。最后一个按钮是远程（Remote），它通常用于调试手机端、桌面端等远程的应用。

关于更详细的 Redux 开发者工具的用法，请参考官网文档（https://github.com/zalmoxisus/redux-devtools-extension），这里不再赘述。

Redux 并不“低效”

在学习本章以前，你可能会认为 Redux 的开发效率非常低，改变状态需要经历 action 和 reducer 才能完成。其实，Redux 如此良苦用心，是为了在前端搭建一个类似 Github 的版本管理器，任何时候发生任何变化都会被记录在案，这不仅让撤销/重做的实现变得非常容易，而且借助 Redux 开发者工具，还可以实现 DIFF 分析和时间旅行等炫酷的功能。

当然，Redux 的优点一言难尽，除了清晰的状态管理之外，全局单一的数据源、action 的集中管控、中间件的拓展能力以及单向数据流等都是非常优秀的设计。

Redux 并不“低效”，它帮助 React 分担了管理数据的工作，让 React 可以专心负责渲染，同时也给我们带来了非常好的开发体验。

总结

- 使用 Redux Undo 的高阶函数配置 reducer，就可以实现撤销/重做功能。
- Redux 开发者工具提供了丰富的功能，不仅提供了日志监控（Log monitor）、检测器（Inspector）和图表（Chart）三个界面来帮助开发者实时追踪程序中 action 和 state 的每一个变化，还在底部工具栏里提供了 action 发起器（Dispatcher）、时间滑块（Slider）、导入/导出（Import/Export）、远程（Remote）等多个按钮来实现手动发起 action、时间旅行调试、导入/导出程序状态、远程调试等炫酷的功能。
- Redux 并不低效，它给我们带来了清晰的状态管理和非常好的开发体验。

第13章 测试

测试在软件工程中是非常重要的一个环节，它不仅能用来做质量保证，还可以用来表达设计或作为文档记录。通常情况下，测试的成本是很高的，不过因为在 React 与 Redux 的应用中大多是一些小型的函数和组件，所以编写测试会相对简单一些。本章将会通过一个例子演示如何在 React 与 Redux 应用中进行测试，测试的内容包括：发起 action 创建函数是否可以正确地创建 action 对象，组件和容器的渲染结果是否正确，模拟用户单击按钮后，组件中的函数是否被正确调用，reducer 纯函数的输入/输出是否正确等。

例子

本章例子编写了四个测试脚本，分别用于测试计数器程序中的 action 创建函数、reducer 纯函数、组件和容器。

运行方法：

```
npm install
npm test
npm run test:watch
```

测试工具

工欲善其事，必先利其器。在编写测试前，需要先了解一些实用的测试工具，包括 Mocha、Enzyme 和 Expect。下面几节将会详细介绍这三个工具各自的功能以及使用方法。

Mocha

Mocha 是一个具有丰富特性的 JavaScript 测试框架。开发者只需要在 Mocha 提供的全局函数（比如describe、it）中编写测试并运行，就可以生成精美的测试报告。如果测试脚本中包含异常错误，Mocha 还会将错误堆栈（Error Stack）输出到测试报告中，方便开发者定位修复。

一个简单的测试脚本就像这样：

```
describe('reducers', () => {
  describe('counter', () => {
    it('should handle initial state', () => {
      expect(counter(undefined, {})).toBe(0);
    });
  });
});
```

describe和it是 Mocha 测试脚本中最核心的两个全局函数，可以在测试中直接使用。它们的功能和用法介绍如下；

- describe函数被用来包裹一组相关的测试。它是一个函数，第一个参数是这组测试的描述，第二个参数是一个实际执行的函数。
- it函数则用于包裹一个单独的测试。它也是一个函数，第一个参数是名称，第二个参数是一个实际执行的函数。

在运行测试脚本后，describe的第一个参数将作为测试报告的一个小标题，it的第一个参数将作为测试报告的某个测试的名称。比如，上述测试脚本最终生成的测试报告就像这样：

```
reducers
    counter
      ✓ should handle initial state
      ...
```

除了使用describe和it打包测试代码并生成测试报告之外，本章的程序还会使用 Mocha 做这些事情：

- 在编写测试脚本时，使用回调函数done来实现异步测试。
- 在运行测试脚本时，使用这些参数来实现需要的功能：
 - 使用--recursive参数来实现递归运行目录中的所有测试脚本。
 - 使用--compilers js:babel-register参数来编译运行使用 JSX 和 ES2015 语法的测试脚本。
 - 使用--require来加载依赖。
 - 使用--watch来开启监视功能。

当然，Mocha 的功能远不止这些。更多的 Mocha 用法，可以参考官网文档（https://mochajs.org/）。

Enzyme

一个 React 组件所具备的功能主要是渲染界面和提供操作。虽然可以先在浏览器中渲染指定组件，人工进行测试，但是我们希望使用工具将这个过程自动化。为了实现这一点，这个模拟用户交互的自动化工具至少应该具备以下几个功能：

- 获取组件生成的 DOM 节点（或类似 DOM 节点的对象）。
- 获取 DOM 节点的属性，比如获取 p 元素的文本内容。
- 模拟用户操作，触发 DOM 节点事件，比如点击按钮。

Enzyme 作为一个专为 React 设计的 JavaScript 测试工具，就可以实现这些功能。值得一提的是，Enzyme 操作 DOM 的 API 是模仿 jQuery 的 API 设计的，因此非常直观灵活。使用 Enzyme 渲染组件，并操作组件中 DOM 的代码，就像下面这样：

```
import { shallow } from 'enzyme';
import Counter from '../../components/Counter';

// 渲染组件
const component = shallow(
  <Counter counter={counter} {...actions} />
);
```

```
// 获取组件中的button和p元素，并获取p元素的文本
const buttons = component.find('button');
const p = component.find('p');
const text = p.text();

// 模拟用户点击按钮
buttons.at(0).simulate('click');
buttons.at(1).simulate('click');
```

shallow()函数来自 Enzyme，它可以“浅渲染”（渲染深度仅一层，而非递归渲染整个 DOM）出一个ShallowWapper对象。这个ShallowWapper对象拥有丰富的 API，让测试 React 组件变得简单。本例使用了这些 API：

- find()可以找到满足条件的节点（返回的还是一个ShallowWapper对象）。
- text()返回一个节点中的文本。
- at()返回指定索引的节点。
- simulate()模拟当前节点的一个事件。

Enzyme 的更多用法请参考官网文档（http://airbnb.io/enzyme/）。

在本章的程序中，将使用 Enzyme 操作 React 组件生成的 DOM，以此来模拟用户交互进行自动化测试。

Expect

在讲解 Expect 前，需要先了解一个编程概念——断言（assert）。编写代码时，我们总是会做出一些假设，断言就用于在代码中捕捉这些假设，如果最终结果与假设不成立，会抛出一个错误。

在 JavaScript 中，我们通常会使用一些断言库来进行断言操作。Expect 就是一个断言库，它提供了常用的断言函数。本章的程序使用了 Expect 的这些 API：

- toEqual()用于判断对象是否等价于给定的值。
- toMatch用于判断给定的字符串或对象是否匹配指定的模式。
- toBe用于判断一个对象是否严格等价于指定值，相当于使用===进行判断。

- createSpy()用于创建一个间谍函数（Spy Function）。
- toHaveBeenCalled()用于判断间谍函数是否至少被调用过一次。

在 Expect 中，间谍函数被用来追踪它自己的调用情况。比如，本章程序创建了一些间谍函数，然后将它们传入组件中，当按钮被点击时，我们就可以通过这些间谍函数来来判断按钮的点击是否成功触发指定的函数了。间谍函数的简单用法如下所示：

```
const increment = expect.createSpy(); // 创建间谍函数
increment(); // 调用简单函数
expect(increment).toHaveBeenCalled(); // 判断间谍函数是否被调用
```

更多的 Expect 用法，请参考官网文档（https://github.com/mjackson/expect）。

下面将介绍使用这些工具编写测试脚本的具体方法。

测试 action 创建函数

测试普通的 action 创建函数非常简单，只需要使用断言判断 action 创建函数返回的 action 对象是否和预期一样即可：

test/actions/counter.spec.js

```
import expect from 'expect';
import * as actions from '../../actions';

describe('actions', () => {
  describe('counter', () => {
    it('increment should create increment action', () => {
      expect(actions.increment()).toEqual({ type: actions.INCREMENT_COUNTER });
    });

    it('decrement should create decrement action', () => {
      expect(actions.decrement()).toEqual({ type: actions.DECREMENT_COUNTER });
    });
  });
});
```

不过，在引入 Thunk 中间件后，一些 action 创建函数会先进行一些条件判断和异步操作后再返回 action 对象。为了测试这样的 action 创建函数，需要先使用redux-mock-store这个库模拟一个带有中间件的 store，然后使用这个模拟的 store 去发起 action 创建函数，最后使用断言判断被创建的 action 对象是否与预期一致。所用到的redux-mock-store的 API 包括：

1. 模拟一个带有中间件的 store。

```
import configureStore from 'redux-mock-store';
import thunk from 'redux-thunk';

const middlewares = [thunk];
const mockStore = configureStore(middlewares);
const store = mockStore({ counter: 1 });
```

2. 使用 store 发起 action 创建函数。

```
store.dispatch(actions.incrementIfOdd());
```

3. 使用 store 获取被创建的 action 对象列表。

```
const actions = store.getActions();
```

完整的测试代码如下：

test/actions/counter.spec.js

```
import expect from 'expect';
import configureStore from 'redux-mock-store';
import thunk from 'redux-thunk';
import * as actions from '../../actions';

const middlewares = [thunk];
const mockStore = configureStore(middlewares);

describe('actions', () => {
  describe('counter', () => {
    ...

    it('incrementIfOdd should create increment action', () => {
      const expectedActions = [
        { type: actions.INCREMENT_COUNTER }
      ];
```

```
    const store = mockStore({ counter: 1 });
    store.dispatch(actions.incrementIfOdd());
    expect(store.getActions()).toEqual(expectedActions);
  });

  it('incrementIfOdd shouldnt create increment action if counter is even', () => {
    const expectedActions = [];
    const store = mockStore({ counter: 2 });
    store.dispatch(actions.incrementIfOdd());
    expect(store.getActions()).toEqual(expectedActions);
  });

  it('incrementAsync should create increment action', done => {
    const expectedActions = [
      { type: actions.INCREMENT_COUNTER }
    ];
    const store = mockStore({ counter: 0 });
    store.dispatch(actions.incrementAsync(100));
    setTimeout(() => {
      expect(store.getActions()).toEqual(expectedActions);
      done();
    }, 100);
  });
 });
});
```

上述代码执行了这些测试：

- 在当前counter为 1 的状态下，发起incrementIfOdd这个 action 创建函数，将生成类型为INCREMENT_COUNTER的 action 对象。
- 在当前counter为 2 的状态下，发起incrementIfOdd这个 action 创建函数，不会生成类型为INCREMENT_COUNTER的 action 对象。
- 发起incrementAsync这个 action 创建函数将在 100 ms 后生成类型为INCREMENT_COUNTER的 action 对象。

测试 React 组件

在测试 React 组件前，必须先明确组件的功能。在本章的例子中，我们要测试的组件为 Counter，它实现了以下几个功能：

- 渲染一个 p 元素，并在里面显示数值，该数值的初始值为 0。
- 渲染了四个按钮，单击第一个按钮将调用`increment`函数。
- 单击第二个按钮将调用`decrement`函数。
- 单击第三个按钮将调用`incrementIfOdd`函数。
- 单击第四个按钮将调用`incrementAsync`函数。

为了测试上述功能，需要先编写一个`setup`函数用于生成被测试的组件、action 函数、组件中的四个按钮和 p 元素。

test/components/Counter.spec.js

```
import expect from 'expect';
import React from 'react';
import { shallow } from 'enzyme';
import Counter from '../../components/Counter';

function setup(counter = 0) {
  const actions = {
    increment: expect.createSpy(),
    incrementIfOdd: expect.createSpy(),
    incrementAsync: expect.createSpy(),
    decrement: expect.createSpy()
  };
  const component = shallow(
    <Counter counter={counter} {...actions} />
  );

  return {
    component,
    actions,
    buttons: component.find('button'),
    p: component.find('p')
```

```
  };
}
```

接下来，将使用 Mocha 来运行测试。

```
describe('components', () => {
  describe('Counter', () => {
    it('should display count', () => {
      const { p } = setup();
      expect(p.text()).toMatch(/^Clicked: 0 times/);
    });

    it('first button should call increment', () => {
      const { buttons, actions } = setup();
      buttons.at(0).simulate('click');
      expect(actions.increment).toHaveBeenCalled();
    });

    it('second button should call decrement', () => {
      const { buttons, actions } = setup();
      buttons.at(1).simulate('click');
      expect(actions.decrement).toHaveBeenCalled();
    });

    it('third button should call incrementIfOdd', () => {
      const { buttons, actions } = setup(43);
      buttons.at(2).simulate('click');
      expect(actions.incrementIfOdd).toHaveBeenCalled();
    });

    it('fourth button should call incrementAsync', () => {
      const { buttons, actions } = setup();
      buttons.at(3).simulate('click');
      expect(actions.incrementAsync).toHaveBeenCalled();
    });
  });
});
```

测试容器组件

由于容器组件包裹了展示组件，而且它的输入依赖于 Redux，因此不能再像测试组件那样使用“浅渲染”，而应该使用 Enzyme 的mount方法递归渲染出所有的 DOM。不幸的是，这种完整的 DOM 渲染需要在全局作用域中提供一套完整的 DOM API，也就是说必须在浏览器环境下才能使用mount，而本例的测试却是在 Node.js 环境中进行的。为了解决这个问题，我们需要使用 JSDOM 来在 Node.js 中模拟一个类似浏览器的环境。



JSDOM 是 DOM 和 HTML 标准的 JavaScript 实现，用于在 Node.js 中模拟浏览器环境。

推荐在所有的测试脚本运行前使用 JSDOM，这样可以保证所有的测试脚本中的 React 组件都享受到“浏览器环境”的待遇。

test/setup.js

```
import { jsdom } from 'jsdom';

global.document = jsdom('<!doctype html><html><body></body></html>');
global.window = document.defaultView;
global.navigator = global.window.navigator;
```

接下来，就可以使用mount函数进行完整的 DOM 渲染了。

test/containers/App.spec.js

```
import expect from 'expect';
import React from 'react';
import { mount } from 'enzyme';
import { Provider } from 'react-redux';
import App from '../../containers/App';
import configureStore from '../../store/configureStore';

function setup(initialState) {
  const store = configureStore(initialState);
  const app = mount(
    <Provider store={store}>
      <App />
```

```
    </Provider>
  );
  return {
    app,
    buttons: app.find('button'),
    p: app.find('p')
  };
}

describe('containers', () => {
  describe('App', () => {
    it('should display initial count', () => {
      const { p } = setup();
      expect(p.text()).toMatch(/^Clicked: 0 times/);
    });

    it('should display updated count after increment button click', () => {
      const { buttons, p } = setup();
      buttons.at(0).simulate('click');
      expect(p.text()).toMatch(/^Clicked: 1 times/);
    });

    it('should display updated count after decrement button click', () => {
      const { buttons, p } = setup();
      buttons.at(1).simulate('click');
      expect(p.text()).toMatch(/^Clicked: -1 times/);
    });

    it('shouldnt change if even and if odd button clicked', () => {
      const { buttons, p } = setup();
      buttons.at(2).simulate('click');
      expect(p.text()).toMatch(/^Clicked: 0 times/);
    });

    it('should change if odd and if odd button clicked', () => {
      const { buttons, p } = setup({ counter: 1 });
      buttons.at(2).simulate('click');
      expect(p.text()).toMatch(/^Clicked: 2 times/);
```

```
    });
  });
});
```

测试 reducer 纯函数

因为 reducer 是纯函数，所以测试 reducer 非常简单，只需要使用断言判断输入/输出即可：

test/reducers/counter.spec.js

```
import expect from 'expect';
import counter from '../../reducers/counter';
import { INCREMENT_COUNTER, DECREMENT_COUNTER } from '../../actions';

describe('reducers', () => {
  describe('counter', () => {
    it('should handle initial state', () => {
      expect(counter(undefined, {})).toBe(0);
    });

    it('should handle INCREMENT_COUNTER', () => {
      expect(counter(1, { type: INCREMENT_COUNTER })).toBe(2);
    });

    it('should handle DECREMENT_COUNTER', () => {
      expect(counter(1, { type: DECREMENT_COUNTER })).toBe(0);
    });

    it('should handle unknown action type', () => {
      expect(counter(1, { type: 'unknown' })).toBe(1);
    });
  });
});
```

运行测试脚本

编写完上述几个测试脚本，接下来在 package.json 中编写运行测试脚本的命令：

package.json

```
"test": "cross-env NODE_ENV=test mocha --recursive --compilers js:babel-register --
    require ./test/setup.js",
 "test:watch": "npm test -- --watch"
```

cross-env是个微型库，可以让你在使用 UNIX 风格的写法来设置环境类型（NODE_ENV=test），但它同样可以在 Windows 中运行。注意，NODE_ENV=test在本程序中可有可无，但是在真实项目中，我们通常需要根据环境变量做一些特殊的设置，所以这里提前写好，为后期拓展做准备。

Mocha 默认运行 test 子目录里面的测试脚本。所以，一般都会把测试脚本放在 test 目录下。另外这里还使用了 Mocha 的几个运行参数：

- --recursive参数表示 test 子目录下面所有的测试用例，不管在哪一层，都会执行。
- --compilers js:babel-register参数表示使用babel-register来编译测试脚本，因为我们的脚本是使用 ES2015 和 JSX 编写的。
- --require参数用于加载依赖，这里加载的是./test/setup.js，用于模拟一个拥有完整 DOM API 的浏览器环境。
- --watch参数用于开启 Mocha 的监视功能，一旦代码发生变动，将会自动运行所有测试脚本。

在命令行输入：

```
npm test
```

运行结果：

```
actions
    counter
      ✓ increment should create increment action
      ✓ decrement should create decrement action
      ✓ incrementIfOdd should create increment action
      ✓ incrementIfOdd shouldnt create increment action if counter is even
```

```
    ✓ incrementAsync should create increment action (101ms)

components
  Counter
    ✓ should display count
    ✓ first button should call increment
    ✓ second button should call decrement
    ✓ third button should call incrementIfOdd
    ✓ fourth button should call incrementAsync

containers
  App
    ✓ should display initial count
    ✓ should display updated count after increment button click
    ✓ should display updated count after decrement button click
    ✓ shouldnt change if even and if odd button clicked
    ✓ should change if odd and if odd button clicked

reducers
  counter
    ✓ should handle initial state
    ✓ should handle INCREMENT_COUNTER
    ✓ should handle DECREMENT_COUNTER
    ✓ should handle unknown action type


19 passing (230ms)
```

注意，虽然本例将describe的标题设计成与测试脚本目录一一对应的结构，但不是非这么做不可。对describe标题来说，唯一的要求就是要有较好的可读性。

至此，整个测试工作就完成了。

总结

- Mocha 是一个具有丰富特性的 JavaScript 测试框架。开发者只需要根据规则在 Mocha 提供的全局函数（比如describe、it）中编写测试并运行，就可以生成精美的测试报

告。如果测试脚本中包含异常错误，Mocha 还会将错误堆栈输出到测试报告中，方便开发者定位和修复。

- Enzyme 是一个专为 React 设计的 JavaScript 测试工具，它让测试 React 组件变得更简单。
- Expect 是个断言库，提供了常用的断言函数。

第14章

Redux的全局状态与React组件的内部状态

一个 Web 应用中的状态是非常繁杂的，不仅包括 UI 的状态，还有服务器响应、缓存数据、本地生成尚未持久化到服务器的数据等。那么应该将这些状态放在哪里来管理呢？是 Redux 的全局状态还是 React 组件的内部状态？本章将会通过一个实例来解答这个问题。

例子

本章例子是 Redux 的官方示例 todomvc，包含增/删/改以及过滤显示等功能（见图 14-1）。

全局状态与内部状态的定义

Redux 的全局状态就是通过`store.getState()`获取的 state。因为这个 state 是全局唯一的，所以我们称其为“全局状态”。

React 组件的内部状态就是通过 `this.state` 获取的 state（这里的`this` 指的是组件实例）。因为这个 state 通常只在组件内部使用，所以我们称其为“内部状态”。

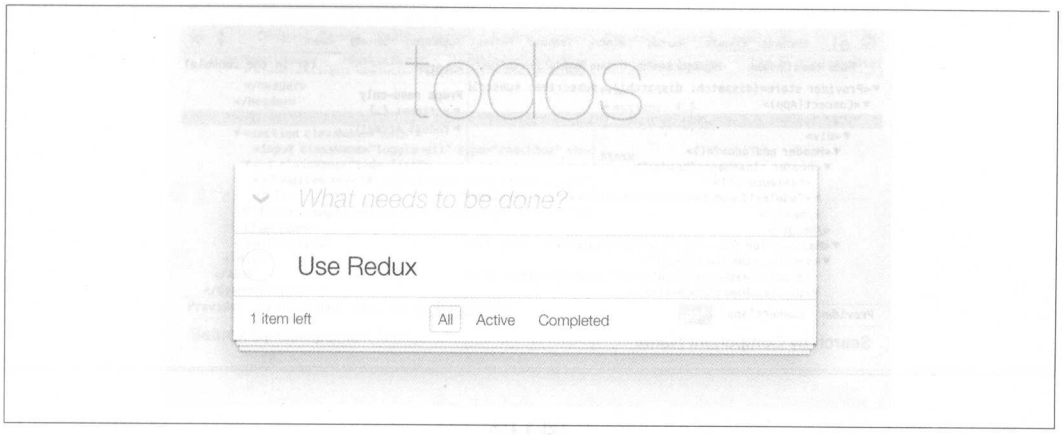


图 14-1

本章为了便于区分两种 state，采用了“全局状态”和“内部状态”这两个名称。在后面的章节中，Redux 的 state 将使用“state”来命名，因为使用“全局状态”或“状态”容易引起歧义。React 组件的 state 则继续使用“内部状态”这个名称。

全局状态完全替代了内部状态?

前面章节的计数器程序中没有用到组件的内部状态，而是通过 Redux 的全局状态来管理数据的。那么既然 Redux 提供了一个全局状态，是不是就不需要组件的内部状态了呢？所有组件都可以写为无状态函数了吗？

让我们来看看 todomvc 是如何处理全局状态和内部状态的。运行程序，打开 React 开发者工具，可以发现：

App 组件没有内部状态（见图 14-2）。

Header 组件没有内部状态（见图 14-3）。

TodoTextInput 组件有内部状态（见图 14-4）。

MainSection 组件有内部状态（见图 14-5）。

TodoItem 组件有内部状态（见图 14-6）。

Footer 组件没有内部状态（见图 14-7）。

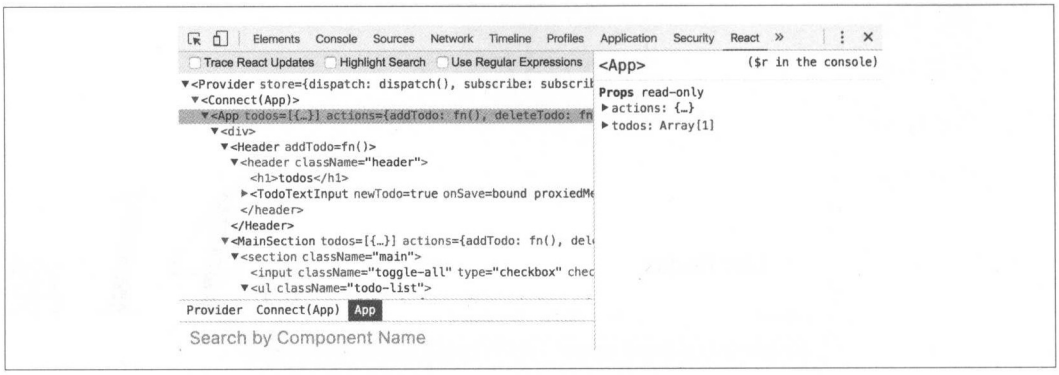


图 14-2



图 14-3

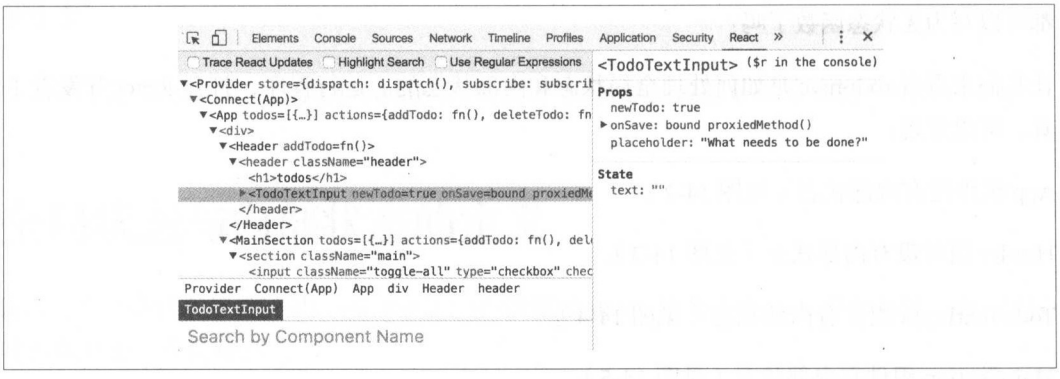


图 14-4

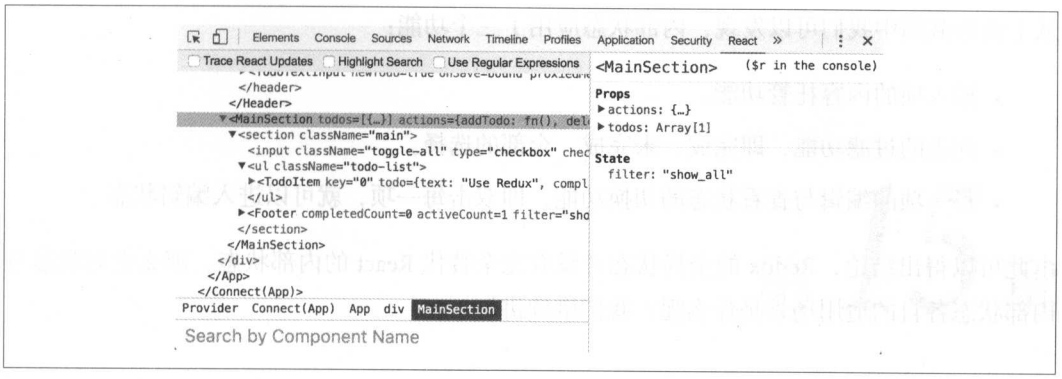


图 14-5

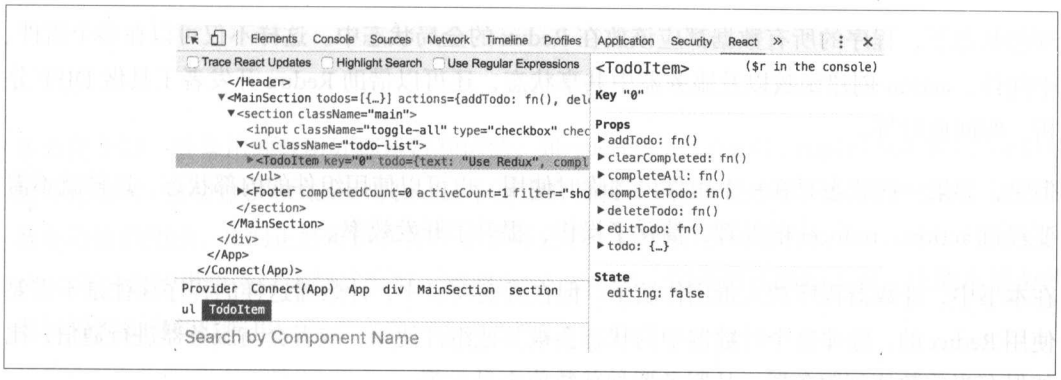


图 14-6

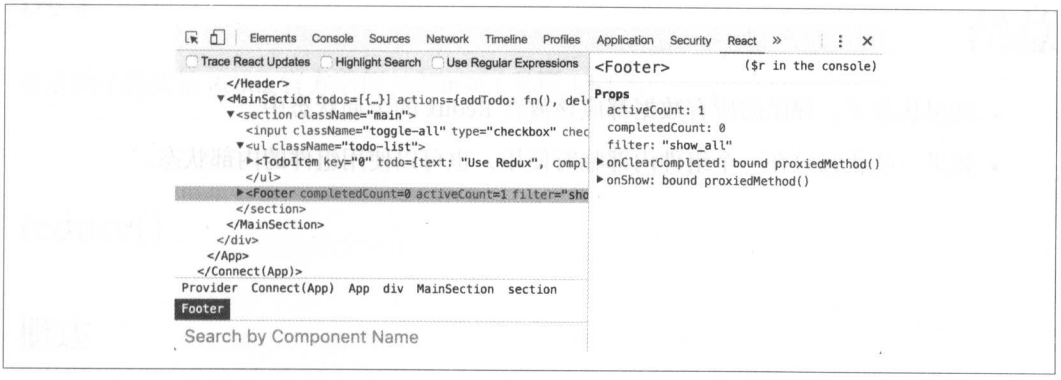


图 14-7

从上面的截图中我们可以发现，内部状态应用了三个功能：

- 输入项的内容托管功能。
- 列表的过滤功能，即完成、未完成、全部的选择。
- 每一项的编辑与查看状态的切换功能，即双击每一项，就可以进入编辑状态。

由此可以得出结论，Redux 的全局状态并没有完全替代 React 的内部状态。那么全局状态与内部状态各自的适用场景是什么呢？我们继续讲解。

全局状态与内部状态的适用场景

理想状态下，程序的所有数据都应该放在 Redux 的全局状态中。这样不仅可以在多个组件、中间件、action 创建函数以及服务器中共享状态，还可以借助 Redux 开发者工具做 DIFF 分析、时间旅行等。

但是，如果一些状态只在一个组件内部临时使用，也可以使用组件的内部状态，这样就不需要经过 action、reducer 的处理，简化了操作，提升了开发效率。

在本书中，计数器程序被大量用作演示。但在真实项目中，计数器这样的程序往往是不需要使用 Redux 的，除非这个计数器中的状态会被其他组件使用，或者会与服务器进行通信，比如保存当前数值到服务器、从服务器加载数值来显示等。

总结

- 理想状态下，程序的所有数据都应该放在 Redux 的全局状态中。
- 如果一些状态只在一个组件内部临时使用，也可以使用组件的内部状态。

第15章

React 与 Redux 中的数组处理

本章将介绍一些常用的数组处理函数和语法，如reduce()、filter()、map()、every()、some()、展开运算符。这些知识和 React 与 Redux 本身没有直接关系，但是本章的示例中包含了这些函数和语法的用法，我们正好可以在程序中学习它们。另外值得一提的是，reduce()、filter()、map()等函数的思想来自函数式编程，感谢 JavaScript 是一门函数式编程语言，让原本复杂的逻辑处理变得如此简单。

例子

本章例子仍然是 Redux 官方示例 todomvc（见图 15-1）。

reduce()

概述

reduce()方法接收一个函数作为累加器（accumulator），数组中的每个值（从左到右）开始合并，最终为一个值。

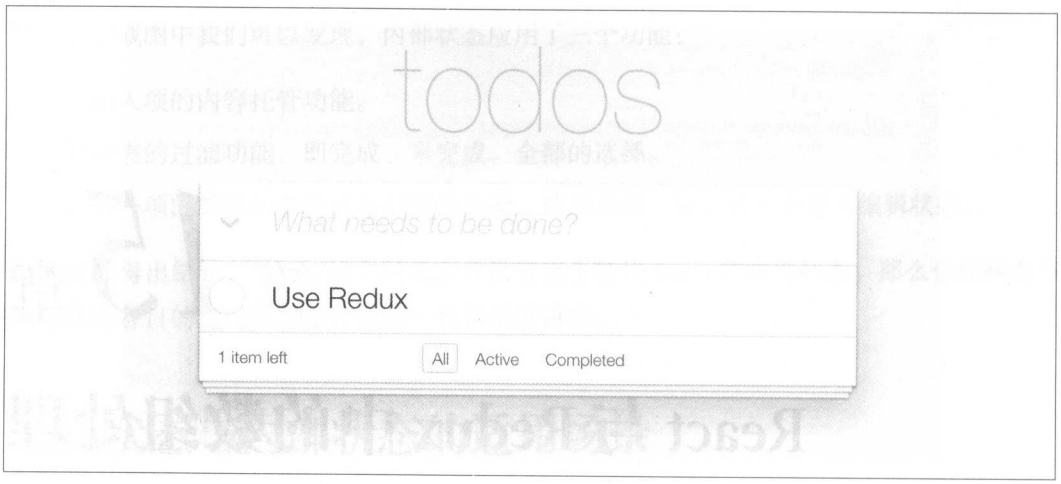


图 15-1

语法

```
array.reduce(callback,[initialValue])
```

参数解释如下。

- callback：执行数组中每个值的函数（也可以叫reducer），包含四个参数。
 - previousValue：上一次调用回调返回的值，或者是提供的初始值（initialValue）。
 - currentValue：数组中当前被处理的元素。
 - index：当前元素在数组中的索引。
 - array：调用reduce的数组
- initialValue：作为第一次调用callback的第一个参数。这个参数不是必需的。

示例

components/MainSection.js 第 69 行

```
const completedCount = todos.reduce(
      (count, todo) => (todo.completed ? count + 1 : count),
      0);
```

代码解释：

- todos是一个数组。
- reduce()的第一个参数是箭头函数，这个箭头函数的第一个参数count是上一个返回值，第二个参数todo是当前元素的值。
- reduce()的第二个参数0是个初始化值，作用是使count 的初始值为0 。

遍历数组todos的第一个值的时候，count为0，todo是todos的第一项，返回值加一或者不变（如果todo.completed为true，则返回count + 1，否则返回count）。

遍历数组todos的第二个值的时候，count为上一个返回值，todo是todos的第二项，返回值加一或者不变。

……

遍历结束后，即可得到数组中completed属性为true的个数，也就是已完成的任务的个数。

filter()

概述

filter()方法使用指定的函数测试所有元素，并创建一个包含所有通过测试的元素的新数组。

语法

```
array.filter(callback[, thisArg])
```

参数解释如下。

- callback：用来测试数组的每个元素的函数，包含三个参数。返回 true 表示保留该元素（通过测试），返回 false 则不保留。
 - currentValue：数组中当前被传递的元素。
 - index：元素的索引。
 - array：被遍历的数组。
- thisArg：可选。执行callback时使用的 this 值。

示例

reducers/todos.js 第 27 行

```
return state.filter(todo =>
        todo.id !== action.id
      )
```

代码解释：

- state是个任务数组。
- filter() 的参数是箭头函数。该箭头函数只有一个参数todo，也就是数组的当前项元素，箭头后面那个判断语句，如果返回true则保留当前项，反之则移除当前项。

该代码段的作用是，过滤掉任务数组中id与指定id相同的任务，返回一个新的任务数组。

map()

概述

map()方法返回一个由原数组中的每个元素调用一个指定方法后的返回值组成的新数组。

语法

```
array.map(callback[, thisArg])
```

参数解释如下。

- callback：原数组中的元素调用该方法后返回一个新的元素。该方法包含如下三个参数。
 - currentValue：数组中当前被传递的元素。
 - index：元素的索引。
 - array：被遍历的数组。
- thisArg：可选。执行callback时使用的 this 值。

示例

reducers/todos.js 第 32 行

```
return state.map(todo => (
        todo.id === action.id ?
          Object.assign({}, todo, { text: action.text }) :
          todo
      ));
```

代码解释：

- state 是变化前的 state，是一个数组。
- map() 的参数是个箭头函数。该箭头函数的第一个参数是todo，返回值是一个使用三元运算符的表达式，用于返回新元素。如果id匹配，则通过Object.assign()合并一个新的属性，也就是给todo添加或者重写一个text属性，属性值为action.text。



Object.assign() 方法可以把任意数目的源对象自身的可枚举属性拷贝给目标对象，然后返回目标对象。语法为Object.assign(target, ...sources)，target是目标对象，sources是任意数目的源对象。

这段代码的作用是为数组中指定的任务更新text属性。

every()

概述

every()方法用于测试数组中所有元素是否都通过了指定函数的测试。

语法

```
array.every(callback[, thisArg])
```

参数解释如下。

- callback：用来测试每个元素的函数。
 - currentValue：数组中当前被传递的元素。
 - index：元素的索引。
 - array：被遍历的数组。
- thisArg：可选。执行callback时使用的 this 值。

示例

reducers/todos.js 第 46 行

```
const areAllMarked = state.every(todo => todo.completed)
```

遍历任务数组，每一项任务的completed属性均为true时候，返回true。

some()

概述

some()方法用于测试数组中是否至少有一项元素通过了指定函数的测试。

语法

```
array.some(callback[, thisArg])
```

参数解释如下。

- callback：用来测试每个元素的函数。
 - currentValue：数组中当前被传递的元素。
 - index：元素的索引。
 - array：被遍历的数组。
- thisArg：可选。执行callback时使用的 this 值。

示例

some() 在本程序中没有用到，可以自己在 Chrome 的控制台中尝试！

展开运算符

概述

展开运算符允许一个表达式在某处展开。常用的场景包括：函数参数、数组元素、解构赋值。另外，JSX 中的组件 props 也可以使用展开运算符来赋值。



解构赋值（destructuring assignment）语法是一个 JavaScript 表达式，它使得从数组或者对象中提取数据赋值给不同的变量成为可能。

语法

用于函数参数:

```
myFunction(...iterableObj);
```

用于数组元素:

```
[...iterableObj, 4, 5, 6]
```

用于解构赋值:

```
const [a, b, ...rest] = [1, 2, 3, 4, 5]
const {a, b, ...rest} = {a:1, b:2, c:3, d:4}
```

用于 React 组件的 props：

```
<App ...iterableObj />
```

示例

展开state数组的每一项到当前的数组。

reducers/todos.js 第 23 行

```
    return [
      {
        id: state.reduce((maxId, todo) => Math.max(todo.id, maxId), -1) + 1,
        completed: false,
        text: action.text,
      },
      ...state,
    ];
```

展开actions的每一个属性到组件中。

components/MainSection.js 第 78 行

```
<TodoItem key={todo.id} todo={todo} {...actions} />
```

总结

- reduce()方法接收一个函数作为累加器（accumulator），数组中的每个值（从左到右）开始合并，最终为一个值。
- filter()方法使用指定的函数测试所有元素，并创建一个包含所有通过测试的元素的新数组。
- map()方法返回一个由原数组中的每个元素调用一个指定方法后的返回值组成的新数组。
- every()方法测试数组的所有元素是否都通过了指定函数的测试。
- some()方法用于测试数组中是否至少有一项元素通过了指定函数的测试。
- 展开运算符允许一个表达式在某处展开。常用的场景包括：函数参数、数组元素、解构赋值。另外，JSX 中的组件 props 也可以使用展开运算符来赋值。

第16章

Redux的大舞台：异步

本章先讲解 JavaScript 事件驱动编程和处理异步的优秀方案——Promise，然后介绍 Redux 中的异步实现，同时还会穿插对 state tree 结构的讲解，最后演示如何使用 Redux 日志中间件追踪程序中的异步和 state 变化。读完本章，希望你不仅能够对异步本身有更深入的了解，而且能感受到 Redux 在处理异步与管理状态方面的巨大优势。

例子

本章例子仍是 Redux 官方的例子，主要是从 Reddit 中请求数据来显示，可以切换 react 和 frontend 关键词来请求不同的数据，还可以刷新当前的数据（见图 16-1）。

JavaScript 事件驱动编程

在讲解 Redux 中的异步实现之前，需要先了解一些 JavaScript 事件驱动编程（Event-driven Programming）的知识。熟悉事件驱动编程才能更好地理解 JavaScript 的异步工作原理。

JavaScript 是一门事件驱动编程语言，如果为特定事件注册了一段代码，这段代码将会在事件被触发时执行。正是事件驱动这个特性让 JavaScript 可以执行异步代码，而不会阻塞后面程序的运行。

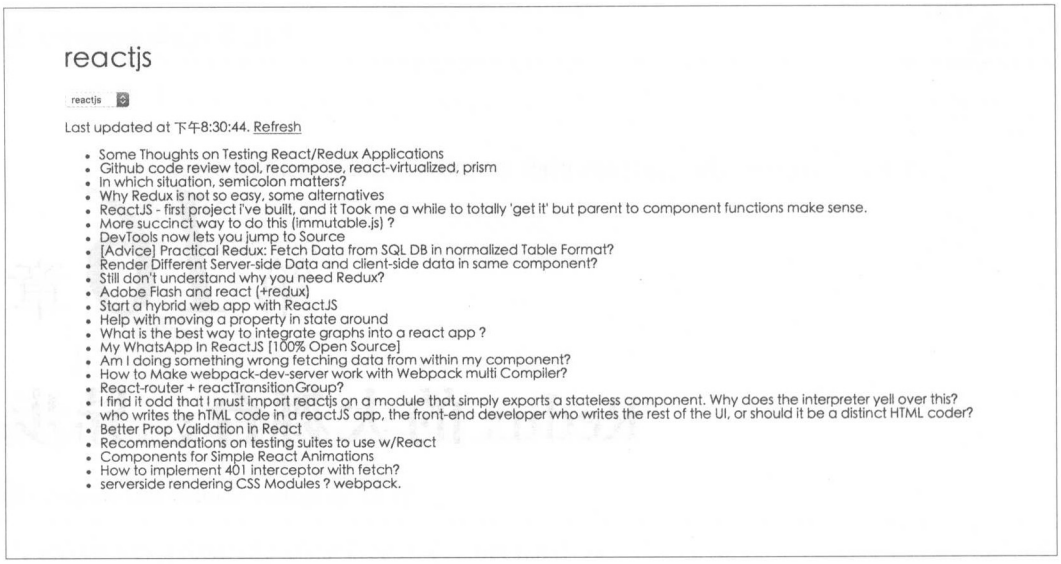


图 16-1

那么，事件的来源都有哪些呢？在前面的计数器程序中，对按钮的单击和setTimeout函数都会触发事件。除此之外，网络请求、磁盘读写等也都可以是事件的来源。JavaScript 为了处理这些事件，使用了事件循环（Event Loop），如图 16-2 所示。

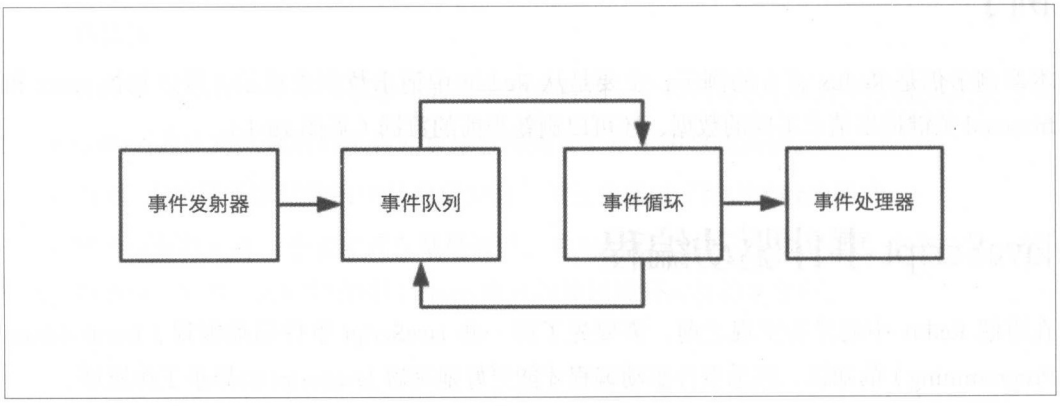


图 16-2

接下来将以前面章节的计数器程序为例，介绍事件驱动的流程。

- 当单击“加一”按钮时，“单击”这个动作作为事件发射器（Event Emitter）给事件队列中添加了一个事件，事件循环从事件队列（Event Queue）中捕捉到这个事件，然后执行注册在这个事件上的代码（通常是个回调函数），也就是事件处理器（Event Handler）。
- 当单击“异步加一”按钮时，“单击”这个动作给事件队列中添加了一个事件，事件循环从事件队列中捕捉到这个事件，然后执行注册在这个事件上的事件处理器。事件处理器中包含一个setTimeout函数，延迟时间过了以后，setTimeout函数向事件队列中添加一个事件，事件循环从事件队列中捕捉到这个事件，然后执行注册在这个事件上的事件处理器。
- 当连续单击“异步加一”按钮时，多个事件被添加到事件队列中，事件循环按顺序捕捉它们，并执行它们的事件处理器。

整个事件驱动的流程就是这么实现的。尝试运行这段代码，体会事件驱动的流程：

```
setTimeout(() => console.log('a'), 0);
console.log('b')
```

由于事件循环必须按事件队列的先后顺序执行，因此console.log('a')必须等到console.log('b')所在的事件执行完成后才能执行。

Promise

得益于事件驱动编程，单线程的 JavaScript 可以将耗时的 I/O 等操作跳过，当这些操作完成后在事件队列中添加一个事件，等待事件循环来捕捉，最后执行注册在该事件上的代码（通常是一个回调函数）。这一切都没有问题，但是随着项目越来越复杂，回调金字塔会让代码变得难以维护，这时使用 Promise 处理异步便是一个不错的选择。

Promise 是处理异步的优秀方案。它不仅可以通过链式操作帮助我们摆脱回调地狱，还可以在链式操作过程中的任何时刻捕捉异常。下面列举一些常见的 Promise 用法。

1. 实例化一个 Promise 对象。具体方法：使用 Promise 构造函数接受一个函数作为参数，该函数的两个参数分别是resolve方法和reject方法。如果异步操作成功，则用resolve方法将 Promise 对象的状态变为“已解决”（resolved）；如果异步操作失败，则用reject方法将状态变为“被拒绝”（rejected）。

```
const promise = new Promise((resolve, reject) => {
  if (/* 异步操作成功 */){
    resolve(value);
  } else {
    reject(error);
  }
});
```

2. 使用Promise.prototype.then(onFulfilled, onRejected)方法来为 Promise 对象指定已解决和被拒绝情况下触发的回调函数。

```
const p1 = new Promise((resolve, reject) => {
  resolve('成功!');
  // or
  // reject ('错误!');
});

p1.then(
  value => console.log(value), // 成功!
  reason => console.log(reason) // 错误!
);
```

由于Promise.prototype.then(onFulfilled, onRejected)会返回一个 Promise 对象，所以还可以使用链式写法。

```
const p2 = new Promise((resolve, reject) => {
  resolve(1);
});

p2.then(value => {
  console.log(value); // 1
  return value + 1;
}).then(value => console.log(value)); // 2
```

3. 使用Promise.prototype.catch(onRejected)只处理 Promise 对象被拒绝的情况。该方法的行为和调用Promise.prototype.then(undefined, onRejected)相同。

```
const p1 = new Promise((resolve, reject) => {
  resolve('成功!');
});
```

```
p1.then(value => {
  console.log(value); // 成功!
  throw '哦，不!';
}).catch(e => console.log(e)); // 哦，不!
```

4. 使用静态方法Promise.resolve(value)直接返回一个已解决的 Promise 对象。

```
Promise.resolve('Success')
 .then(
   value => console.log(value), // Success
   () => {
     // 不会被调用
   }
 );
```

5. 使用静态方法Promise.reject(reason)直接返回一个被拒绝的 Promise 对象。

```
Promise.reject('Testing static reject')
  .then(() => {
      // 未被调用
    },
    reason => console.log(reason) // Testing static reject
  );

Promise.reject(new Error('fail'))
  .then(() => {
      // 未被调用
    },
    error => console.log(error)  // 堆栈跟踪
  );
```

6. 使用Promise.all(iterable)处理一个可迭代对象。Promise.all(iterable) 方法返回一个 Promise 对象，该 Promise 对象会等iterable参数内的所有 Promise 对象都被解决后再被解决，或在第一个 Promise 对象被拒绝时就被拒绝。

```
const promise = Promise.resolve(3);

Promise.all([true, promise])
  .then(values => console.log(values)); // [true, 3]
```

7. 使用Promise.race(iterable)处理一个可迭代对象。Promise.race(iterable)方法返回一个 Promise 对象，该 Promise 对象会在iterable中的任意一个 Promise 对象被解决或拒绝后，立刻以相同的解决值被解决或以相同的拒绝原因被拒绝。

```
const p1 = new Promise((resolve, reject) => {
    setTimeout(resolve, 500, '一');
});
const p2 = new Promise((resolve, reject) => {
    setTimeout(resolve, 100, '二');
});

Promise.race([p1, p2]).then(value => {
  console.log(value); // 二
  // 两个都解决，但p2更快
});
```

本章的程序使用fetch()方法执行网络请求，fetch 方法会返回一个 Promise 对象，无须我们自己创建。fetch()的简单用法如下：

```
fetch(`/data.json`)
  .then(response => response.json())
  .then(json => console.log(json));
```

上述代码的含义如下：

- fetch()方法接受了一个 URL 作为参数，返回了一个 Promise 对象。
- 然后使用then()方法在 Promise 对象中拿到一个 Response 对象，因为这里是 JSON 格式的数据，所以使用response.json()来解析并返回。
- 最后再次使用then()方法拿到上一个then()方法中返回json数据并打印。

关于更详细的 fetch 方法介绍，请参考 Mozilla 开发者网络中的相关介绍（https://developer.mozilla.org/zh-CN/docs/Web/API/Fetch_API/Using_Fetch），这里不再赘述。

Redux 中的异步

了解了 JavaScript 事件驱动编程和处理异步的优秀方法 Promise，接下来将介绍 Redux 中的异步操作。

异步 action 创建函数

Redux 中的异步操作几乎都在 action 创建函数中进行。action 创建函数原本只能实现同步操作，但是可以通过 Thunk 中间件编写异步 action 创建函数。所谓异步 action 创建函数，其实就是包含异步请求的 action 创建函数。本例中的异步 action 创建函数是fetchPosts()。

actions/index.js

```
function fetchPosts(reddit) {
  return dispatch => {
    dispatch(requestPosts(reddit));
    return fetch(`https://www.reddit.com/r/${reddit}.json`)
      .then(response => response.json())
      .then(json => dispatch(receivePosts(reddit, json)));
  };
}
```

这个 action 创建函数之所以返回带有dispatch参数的函数，就是得益于 Thunk 中间件的加工。

异步请求前后的 state 变化

在上述异步 action 创建函数中，我们不仅使用fetch()方法进行了异步请求，还在异步请求的前后分别发起了requestPosts()和receivePosts()这两个 action 创建函数去改变 state。

actions/index.js

```
function requestPosts(reddit) {
  return {
    type: REQUEST_POSTS,
    reddit,
  };
}

function receivePosts(reddit, json) {
  return {
    type: RECEIVE_POSTS,
    reddit,
    posts: json.data.children.map(child => child.data),
```

```
    receivedAt: Date.now(),
  };
}
```

requestPosts()和receivePosts()主要改变了 state tree 中的isFetching和didInvalidate等节点：

reducers/index.js

```
...
case REQUEST_POSTS:
      return Object.assign({}, state, {
        isFetching: true,
        didInvalidate: false,
      });
case RECEIVE_POSTS:
  return Object.assign({}, state, {
    isFetching: false,
    didInvalidate: false,
    items: action.posts,
    lastUpdated: action.receivedAt,
  });
...
```

程序会根据 state tree 中这些节点做一些事情，比如：

1. 我们在异步请求中看到的“Loading...”的显示消失以及文章列表的透明度的变化都是通过isFetching来控制的。

 containers/App.js

   ```
   ...
   const message = isFetching ? <h2>Loading...</h2> : <h2>Empty.</h2>;
   ...
   {isEmpty ? message : <div style={{ opacity: isFetching ? 0.5 : 1 }}>
              <Posts posts={posts} />
            </div>
   ```

2. 另外，还可以通过 state tree 中这些节点来判断是否有必要进行异步请求，从而避免进行不必要的请求以实现优化。判断逻辑是：如果文章数据为空，则应该请求。如果正在获取文章，则不应该请求。如果文章数据失效，则应该请求。最后将其封装

在fetchPostsIfNeeded这个 action 创建函数中。如此一来，就可以使用fetchPostsIfNeeded来替代fetchPosts进行更“智能”的异步请求了。

actions/index.js

```
function shouldFetchPosts(state, reddit) {
  const posts = state.postsByReddit[reddit];
  if (!posts) {
    return true;
  }
  if (posts.isFetching) {
    return false;
  }
  return posts.didInvalidate;
}

export function fetchPostsIfNeeded(reddit) {
  return (dispatch, getState) => {
    if (shouldFetchPosts(getState(), reddit)) {
      return dispatch(fetchPosts(reddit));
    }
    return null;
  };
}
```

事实上，Redux 中的异步操作简单来说就两件事：

- 使用 Thunk 中间件使 action 创建函数可以进行异步操作。
- 在进行异步请求的前后各发起一次 action 去改变 state，进而改变视图和执行一些其他的逻辑。

本例的异步请求只有成功后的处理，在真实项目中，我们应该添加请求失败情况下的处理逻辑。

State tree 结构

上一节中，我们接触了 state tree 中的很多节点。那么 state tree 究竟长什么样？它是如何设计的呢？下面我们一一解答这些问题。

查看 state tree

首先，可以通过 Redux 开发者工具来看看 state tree 的结构。将 Redux 开发者工具切换到 Inspector 界面，单击右边的“State”选项卡，就可以查看 state tree 的结构了。不仅如此，通过单击左侧的 action 列表，还能在右侧查看指定 action 发起后最终生成的 state tree（见图 16-3）。

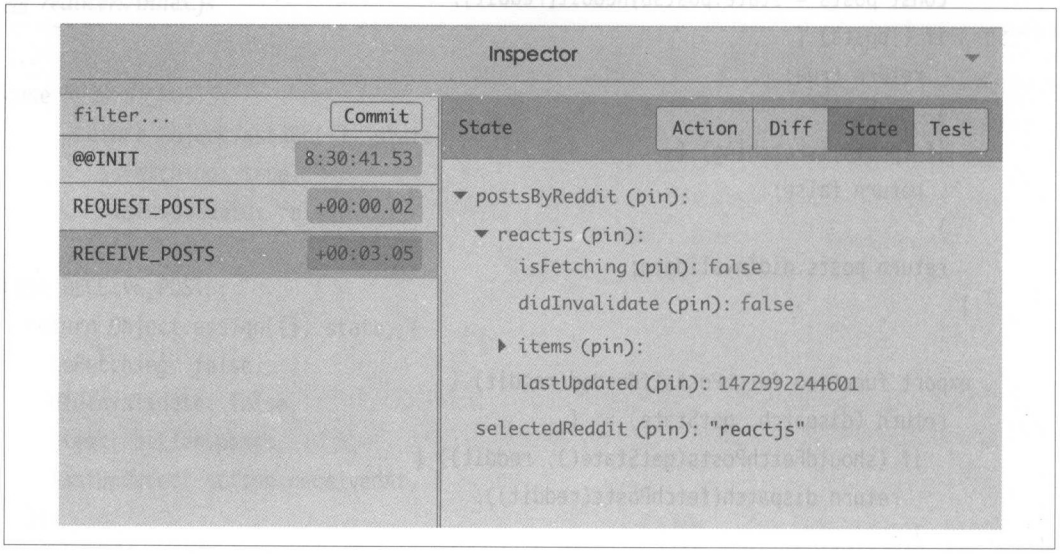


图 16-3

如何构建 state tree

那么 state tree 为什么长这个样子，如何构建 state tree 呢？答案都在 reducer 里面。

reducers/index.js

```
import { combineReducers } from 'redux';
import {
  SELECT_REDDIT, INVALIDATE_REDDIT,
  REQUEST_POSTS, RECEIVE_POSTS,
} from '../actions';

function selectedReddit(state = 'reactjs', action) {
  switch (action.type) {
    case SELECT_REDDIT:
```

```
      return action.reddit;
    default:
      return state;
  }
}

function posts(state = {
  isFetching: false,
  didInvalidate: false,
  items: [],
}, action) {
  switch (action.type) {
    case INVALIDATE_REDDIT:
      return Object.assign({}, state, {
        didInvalidate: true,
      });
    case REQUEST_POSTS:
      return Object.assign({}, state, {
        isFetching: true,
        didInvalidate: false,
      });
    case RECEIVE_POSTS:
      return Object.assign({}, state, {
        isFetching: false,
        didInvalidate: false,
        items: action.posts,
        lastUpdated: action.receivedAt,
      });
    default:
      return state;
  }
}

function postsByReddit(state = { }, action) {
  switch (action.type) {
    case INVALIDATE_REDDIT:
    case RECEIVE_POSTS:
    case REQUEST_POSTS:
```

```
      return Object.assign({}, state, {
        [action.reddit]: posts(state[action.reddit], action),
      });
    default:
      return state;
  }
}

const rootReducer = combineReducers({
  postsByReddit,
  selectedReddit,
});

export default rootReducer;
```

上述代码做了这几件事：

- 写了三个 reducer，postsByReddit 、posts和 selectedReddit。
- 将posts这个 reducer 作为postsByReddit返回值中的一个子节点。
- 使用redux的combineReducers方法把postsByReddit 和 selectedReddit合并起来。



combineReducers方法来自redux，它被用来合并多个 reducer。这些 reducer 被合并后，它们所处理的 state 将会成为 state tree 中的子节点。例如，在前面的计数器程序中，combineReducers({ counter })将会使得计数器的数值从state转移到state.counter中。

因为combineReducers 的合并，所以 state tree 的第一级节点是postsByReddit 和 selectedReddit：

```
{
  postsByReddit,
  selectedReddit,
}
```

而postsByReddit节点的值就是postsByReddit()方法返回的 state：

```
{
   [action.reddit]: posts(state[action.reddit], action),
}
```

[action.reddit]节点的值就是posts()返回的 state：

```
{
  isFetching: false,
  didInvalidate: false,
  items: [],
}
```

现在明白了 state tree 是如何构建了的吧？是的，就是通过 reducer 构建的。

异步与变化

读完了前面的内容，或许你还是一头雾水，其实异步和状态变化都是很简单的概念，但是当我们把它们放在一起时就会让程序变得一团糟。所幸在 Redux 程序中，理清异步与变化的关系比在其他程序中容易得多。下面就让我们分别执行刷新页面、切换关键字、单击“刷新”按钮三个用户操作，来看看分别都触发了哪些函数（包括用户操作所触发的函数和组件生命周期函数），发起了哪些 action，改变了哪些 state。

准备工作

在追踪前，我们需要做三个准备工作：

1. 为 Redux 添加日志中间件 redux-logger。日志中间件可以打印程序运行过程中的action，以及 action 发起前后的 state。

 store/configureStore.js

   ```
   import createLogger from 'redux-logger';
   ...
   applyMiddleware(thunkMiddleware, createLogger())
   ```

 大多数情况下，应该使用 Redux 开发者工具来追踪程序状态变化，此处使用日志中间件是为了在控制台中方便地看出组件生命周期函数的调用和 action 触发时机的时间先后关系。

2. 在componentDidMount和componentWillReceiveProps这两个函数里面打印信息，来追踪它们的触发情况。这两个生命周期函数分别在组件挂载完成后和组件接收新的 props 时被触发。

containers/App.js

```
componentDidMount() {
    console.log('执行componentDidMount');
    ...
  }

  componentWillReceiveProps(nextProps) {
    console.log('执行componentWillReceiveProps', nextProps);
    ...
  }
```

3. 明确各种用户操作所触发的函数和组件生命周期函数中发起了哪些 action 创建函数，这些 action 创建函数又发起了哪些 action 对象。

containers/App.js

```
componentDidMount() {
    ...
    const { dispatch, selectedReddit } = this.props;
    dispatch(fetchPostsIfNeeded(selectedReddit));
  }

  componentWillReceiveProps(nextProps) {
    if (nextProps.selectedReddit !== this.props.selectedReddit) {
      const { dispatch, selectedReddit } = nextProps;
      dispatch(fetchPostsIfNeeded(selectedReddit));
    }
  }
  // 切换下拉框事件的注册函数
  handleChange(nextReddit) {
    this.props.dispatch(selectReddit(nextReddit));
  }
  // 刷新按钮点击事件的注册函数
  handleRefreshClick(e) {
    e.preventDefault();

    const { dispatch, selectedReddit } = this.props;
```

```
    dispatch(invalidateReddit(selectedReddit));
    dispatch(fetchPostsIfNeeded(selectedReddit));
  }
```

actions/index.js

```
import fetch from 'isomorphic-fetch';

export const REQUEST_POSTS = 'REQUEST_POSTS';
export const RECEIVE_POSTS = 'RECEIVE_POSTS';
export const SELECT_REDDIT = 'SELECT_REDDIT';
export const INVALIDATE_REDDIT = 'INVALIDATE_REDDIT';

export function selectReddit(reddit) {
  return {
    type: SELECT_REDDIT,
    reddit,
  };
}

export function invalidateReddit(reddit) {
  return {
    type: INVALIDATE_REDDIT,
    reddit,
  };
}

function requestPosts(reddit) {
  return {
    type: REQUEST_POSTS,
    reddit,
  };
}

function receivePosts(reddit, json) {
  return {
    type: RECEIVE_POSTS,
    reddit,
```

```
    posts: json.data.children.map(child => child.data),
    receivedAt: Date.now(),
  };
}

function fetchPosts(reddit) {
  return dispatch => {
    dispatch(requestPosts(reddit));
    return fetch(`https://www.reddit.com/r/${reddit}.json`)
      .then(response => response.json())
      .then(json => dispatch(receivePosts(reddit, json)));
  };
}

function shouldFetchPosts(state, reddit) {
  const posts = state.postsByReddit[reddit];
  if (!posts) {
    return true;
  }
  if (posts.isFetching) {
    return false;
  }
  return posts.didInvalidate;
}

export function fetchPostsIfNeeded(reddit) {
  return (dispatch, getState) => {
    if (shouldFetchPosts(getState(), reddit)) {
      return dispatch(fetchPosts(reddit));
    }
    return null;
  };
}
```

从上述代码可以看出，组件生命周期函数和用户操作所触发的函数、action 创建函数、action 对象之间的关系如表 16-1 所示。

表 16-1

组件生命周期函数和用户操作所触发的函数	action 创建函数	action 对象
componentDidMount	fetchPostsIfNeeded(selectedReddit)	REQUEST_POSTS、RECEIVE_POSTS
componentWillReceiveProps	fetchPostsIfNeeded(selectedReddit)（在关键字改变时发起）	REQUEST_POSTS、RECEIVE_POSTS
handleChange（切换下拉框）	selectReddit(nextReddit)	SELECT_REDDIT
handleRefreshClick（单击刷新按钮）	invalidateReddit(selectedReddit) fetchPostsIfNeeded(selectedReddit)	INVALIDATE_REDDIT、REQUEST_POSTS、RECEIVE_POSTS

追踪异步与变化

准备工作做完了，然后打开浏览器，执行下面的用户操作，查看打印信息。

1. 刷新页面，打印信息如图 16-4 所示。

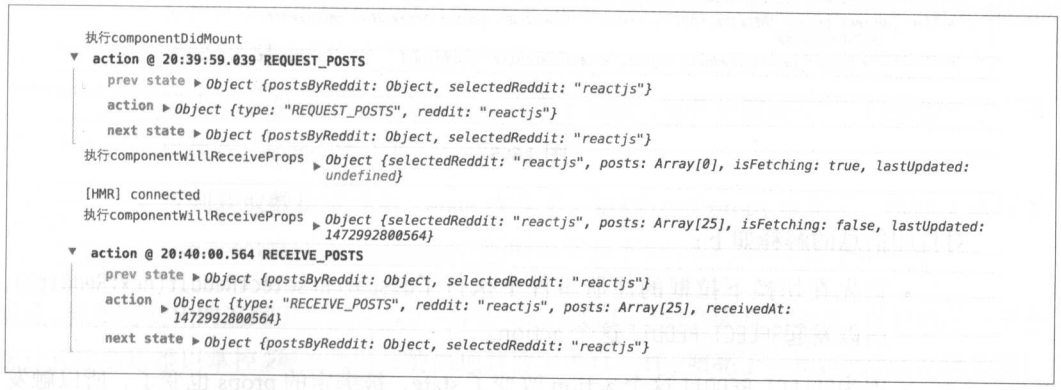


图 16-4

对打印信息的解释如下：

- 刷新页面首先需要渲染组件，所以先执行了componentDidMount，也就是组件挂载完成。

- 因为我们在 componentDidMount 方法中执行了 dispatch(fetchPostsIfNeeded(selectedReddit))，所以又发起REQUEST_POSTS 这个 action。
- 因为REQUEST_POSTS 这个 action 改变了 state，组件中被绑定的 props 也变了，所以执行componentWillReceiveProps。
- 等到获取数据成功后，state 改变了，被绑定的 props 也变了，所以又执行了 componentWillReceiveProps。我们可以看到 posts 里面已经有值，这时触发了 RECEIVE_POSTS 这个 action。

[HMR] 是热替换的意思，这里不详细叙述。

2. 切换关键字，打印信息如图 16-5 所示。

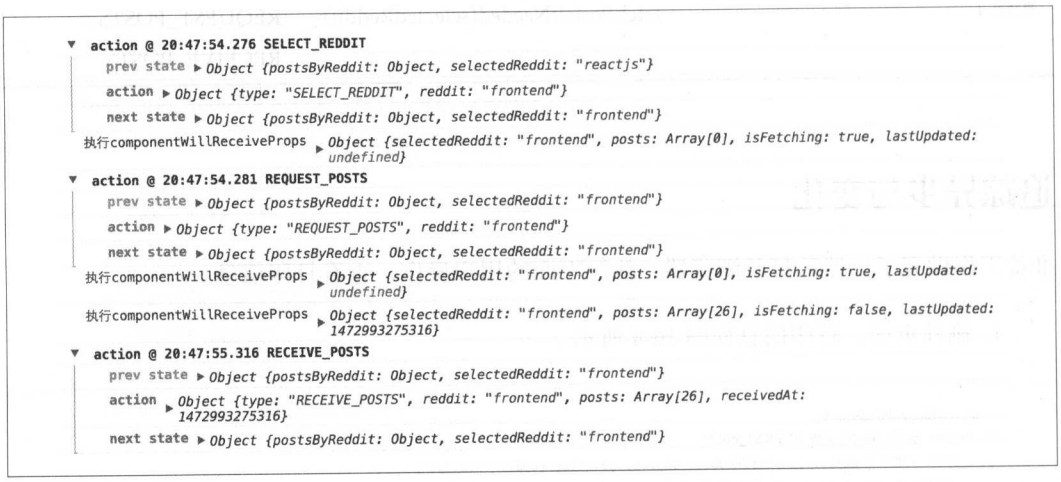


图 16-5

对打印信息的解释如下：

- 首先在切换下拉框的注册事件中执行了dispatch(selectReddit(nextReddit))，所以发起SELECT_REDDIT 这个 action。
- 因为SELECT_REDDIT 这个 action 改变了 state，被绑定的 props 也变了，所以触发了componentWillReceiveProps 。
- 因为在关键词改变时会执行dispatch(fetchPostsIfNeeded(selectedReddit))，所以又发起了REQUEST_POSTS 这个 action。
- 因为REQUEST_POSTS 这个 action 改变了 state，被绑定的 props 也变了，所以又触发了componentWillReceiveProps。

- 等到获取数据成功后，state 改变了，被绑定的 props 也变了，所以又执行了 componentWillReceiveProps，同时也发起了RECEIVE_POSTS 这个 action。

3. 单击“刷新”按钮，打印信息如图 16-6 所示。

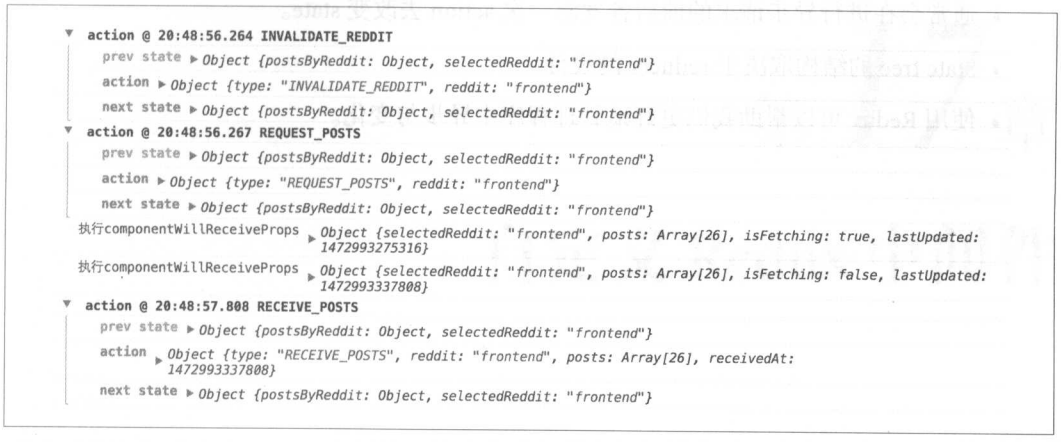


图 16-6

对打印信息的解释如下：

- 首先在刷新按钮的点击事件中执行了dispatch(invalidateReddit(selectedReddit))和dispatch(fetchPostsIfNeeded(selectedReddit))，所以发起INVALIDATE_REDDIT 和REQUEST_POSTS 这个 action。
- 因为REQUEST_POSTS 这个 action 改变了 state，组件中被绑定的 props 也变了，所以执行componentWillReceiveProps。
- 等到获取数据成功后，state 改变了，被绑定的 props 也变了，所以又执行了 componentWillReceiveProps，同时也发起了RECEIVE_POSTS 这个 action。

至此，异步与变化的追踪工作就完成了。异步与变化像是一对会隐身的幽灵，它们想尽一切办法让程序变得难以掌控。可是 Redux 的出现就像宝莲灯一样，照亮了一切，让世间充满光明！

总结

- JavaScript 是一门事件驱动编程语言，事件驱动这个特性让 JavaScript 可以执行异步代码，而不会阻塞后面程序的运行。

- Promise 是处理异步的优秀方案。它不仅可以通过链式操作帮助我们摆脱回调地狱，还可以在链式操作过程中的任何时刻捕捉异常。
- Redux 只能实现同步操作，但是可以通过 Thunk 中间件实现异步。
- 通常会在进行异步请求的前后各发起一次 action 去改变 state。
- State tree 的结构取决于 reducer 的设计。
- 使用 Redux 可以帮助我们更好地管理程序中异步与变化。

第17章

自定义Redux中间件

本章讲解如何自定义 Redux 中间件，具体内容包括中间件的功能、定义、执行等。最后会演示一个 API 中间件的具体编写步骤。

例子

本章例子是 Redux 官方示例程序，实现了获取 Github 用户和仓库的功能（见图 17-1）。

中间件的功能

在前面的章节中，我们多次使用了 Thunk 中间件，它使得 action 创建函数可以返回一个函数进行条件判断或者异步操作。如果没有 Thunk 中间件，action 创建函数只能同步地返回一个 action 对象。所以，中间件的一个很常用的功能就是定制 action 创建函数的返回格式。事实上，因为中间件只是 Redux 发起 action 过程中的一个处理函数，所以你可以使用中间件做任何与 action 相关的事情，比如打印 action 日志等。

中间件的最大特点就是可以多个串联使用，因此中间件是 Redux 中最常见的一种拓展形式。可以在这里（https://github.com/xgrommx/awesome-redux）查看社区为我们提供的第三方中间件。

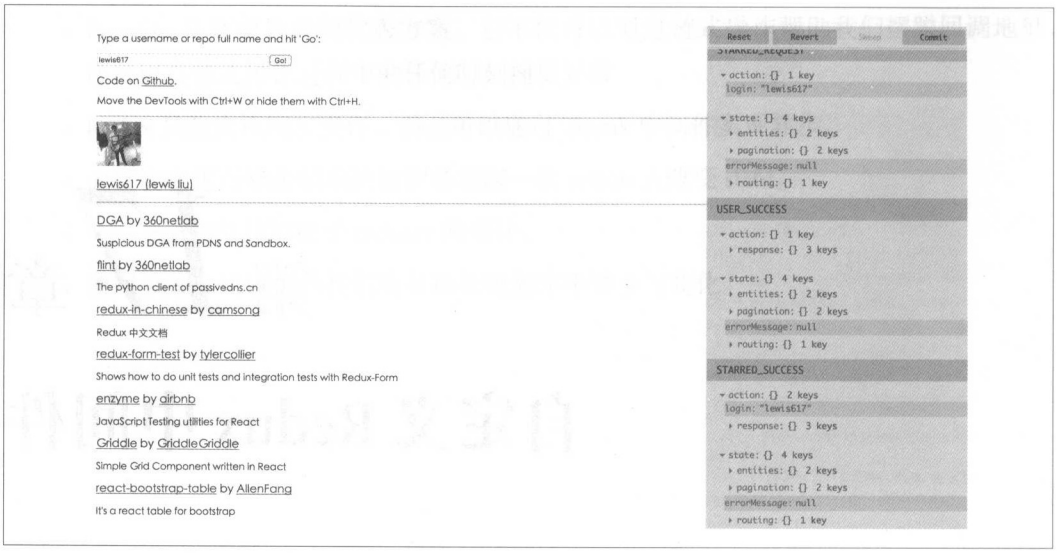


图 17-1

中间件的编写

除了使用别人写好的中间件，编写自定义的中间件也是十分必要的技能。那么如何编写 Redux 中间件呢？非常简单，只需要编写一个嵌套函数：

```
export default store => next => action => {
  return next(action)
};
```

Redux 中间件一共嵌套了三层函数，分别传递了store、next、action这三个参数，并在函数最后返回next(action)。



Redux 中间件的定义使用了函数式编程中的柯里化（Currying）技术。柯里化将一个接收多个参数的函数转化为单参数函数，转化后的函数每次只接收一个参数，然后返回一个新函数，新函数可以继续接收参数，直到接收到所有的参数。

为何不在一层函数中传递三个参数，而要在一层函数中传递一个参数，一共传递三层呢？如果只是为了传递store、next、action这三个参数，当然可以写成一层函数。不过，中间件的

每一层函数将来都会单独运行，所以需要写成嵌套函数，通过运行上级函数返回需要单独运行的函数。每一层函数的意义如下：

- 最外面的函数的参数为 store，这个函数的作用仅仅是为了给每个中间件传递一个 store，使其可以使用 store.getState 和 store.dispatch 这两个方法。
- 中间的函数参数为next ，next 的实质是dispatch 。这层函数实际上是一个高阶函数，它接收一个dispatch函数作为参数，返回了一个更高级的dispatch函数。这个函数将来会以多个首尾相连的方式对dispatch进行一层层的“加工”。
- 最里面的函数的参数为action ，这个函数就是最后要取代dispatch 的函数。

中间件的执行

嵌套函数也是函数，是函数就要运行，JavaScript 使用()来执行一个函数。那么三层嵌套函数的执行当然就是middleware()()()。

- middleware() 返回一个参数为next 的函数，对应中间件的第二层函数。
- middleware()() 返回一个参数为action 的函数，对应中间件的第三层函数，也就是将来要取代dispatch 的函数。
- middleware()()() 相当于执行dispatch 。

中间件的使用是从applyMiddleware 开始的：

```
applyMiddleware(thunk, api)
```

本例一共执行了两个中间件，分别是 thunk 和 api。为了了解内部的执行过程，这里列出 applyMiddleware 的部分源代码：

node_modules/redux/src/applyMiddleware.js

```
import compose from './compose'

...

export default function applyMiddleware(...middlewares) {
  return (createStore) => (reducer, initialState, enhancer) => {
    var store = createStore(reducer, initialState, enhancer)
    var dispatch = store.dispatch
```

```
    var chain = []

    var middlewareAPI = {
      getState: store.getState,
      dispatch: (action) => dispatch(action)
    }
    chain = middlewares.map(middleware => middleware(middlewareAPI))
    dispatch = compose(...chain)(store.dispatch)

    return {
      ...store,
      dispatch
    }
  }
}
```

下面我们来解读上述源代码，并分析中间件中的函数执行情况。

1. 中间件最外层函数的执行发生在这里：

   ```
   chain = middlewares.map(middleware => middleware(middlewareAPI))
   ```

 使用展开语法，将我们写进去的中间件，生成一个中间件数组middlewares。遍历每个中间件，执行最外层的函数，也就是参数为store那一次。这里的store是middlewareAPI：

   ```
   var middlewareAPI = {
         getState: store.getState,
         dispatch: (action) => dispatch(action)
       }
   ```

2. 第二层函数的执行代码是：

   ```
   dispatch = compose(...chain)(store.dispatch)
   ```

 compose是一种函数嵌套的写法，源代码清单就不在此展示了，它的作用是将嵌套的函数写成由逗号隔开的结构。

 chain中的每一项元素都是中间件的第二层函数，将其用逗号隔开放进compose参数里，生成一个嵌套函数。

 store.dispatch是整个嵌套函数的入口参数，它是一个未经任何加工的dispatch，这个可怜的家伙进去后，将被中间件层层加工，经历风雨，变成拥有更多功能的高

级dispatch。这个过程使用了函数式编程中的管道（Pipe）思想，即数据被函数列表依次加工，就像管道一样，最后返回我们想要的结果。

得到加工后的dispatch后，最后使用它重写 store 自带的dispatch：

```
return {
      ...store,
      dispatch
    }
```

3. 第三层函数的执行不在applyMiddleware中。当你使用新的dispatch方法发起 action 时，第三层函数才被执行。

自定义 API 中间件

第 16 章的例子演示了 Redux 处理异步场景的方法，美中不足的是，它没有考虑超时的情况。在真实的开发中，一个异步请求通常需要编写三个 action，分别在开始请求、请求成功和请求失败时被发起。本章的例子不仅包含这三个 action 的处理，还自定义了一个 API 中间件，将请求过程从 action 创建函数转移到自定义的中间件中。如此一来，凡是符合 API 中间件过滤要求的 action 创建函数，都会自动发起相应的请求，并做出处理：

actions/index.js

```
function fetchUser(login) {
  return {
    [CALL_API]: {
      types: [USER_REQUEST, USER_SUCCESS, USER_FAILURE],
      endpoint: `users/${login}`,
      schema: Schemas.USER,
    },
  };
}
```

这个 action 创建函数之所以可以返回一个 key 为[CALL_API] 的对象，就是得益于 API 中间件的定制加工。

其中，CALL_API 来自中间件的export const CALL_API = Symbol('Call API');，它是个 Symbol 类型。Symbol 是一个唯一的、不变的标识，可以用作对象的 key。它是为了避免命名冲突而生的，这里也可以用字符串来代替，不过你就要自己保证命名不发生冲突了。

在中间件处理一系列 action 对象之前，首先要过滤掉不符合指定规则的 action，并做异常处理，也就是消毒：

middleware/api.js

```
const callAPI = action[CALL_API];
  if (typeof callAPI === 'undefined') {
    return next(action);
  }

  let { endpoint } = callAPI;
  const { schema, types } = callAPI;

  if (typeof endpoint === 'function') {
    endpoint = endpoint(store.getState());
  }

  if (typeof endpoint !== 'string') {
    throw new Error('Specify a string endpoint URL.');
  }
  if (!schema) {
    throw new Error('Specify one of the exported Schemas.');
  }
  if (!Array.isArray(types) || types.length !== 3) {
    throw new Error('Expected an array of three action types.');
  }
  if (!types.every(type => typeof type === 'string')) {
    throw new Error('Expected action types to be strings.');
  }
```

接着执行请求，并在开始请求、请求成功或者请求失败后各使用next()发起相应的 action：

```
function actionWith(data) {
    const finalAction = Object.assign({}, action, data);
    delete finalAction[CALL_API];
    return finalAction;
  }
```

```
const [requestType, successType, failureType] = types;
next(actionWith({ type: requestType }));

return callApi(endpoint, schema).then(
  response => next(actionWith({
    response,
    type: successType,
  })),
  error => next(actionWith({
    type: failureType,
    error: error.message || 'Something bad happened',
  }))
);
```

至此，API 中间件就编写完成了。经过 API 中间件的加工，发起携带CALL_API的 action 创建函数就变成了一系列的流程，如图 17-2 所示。

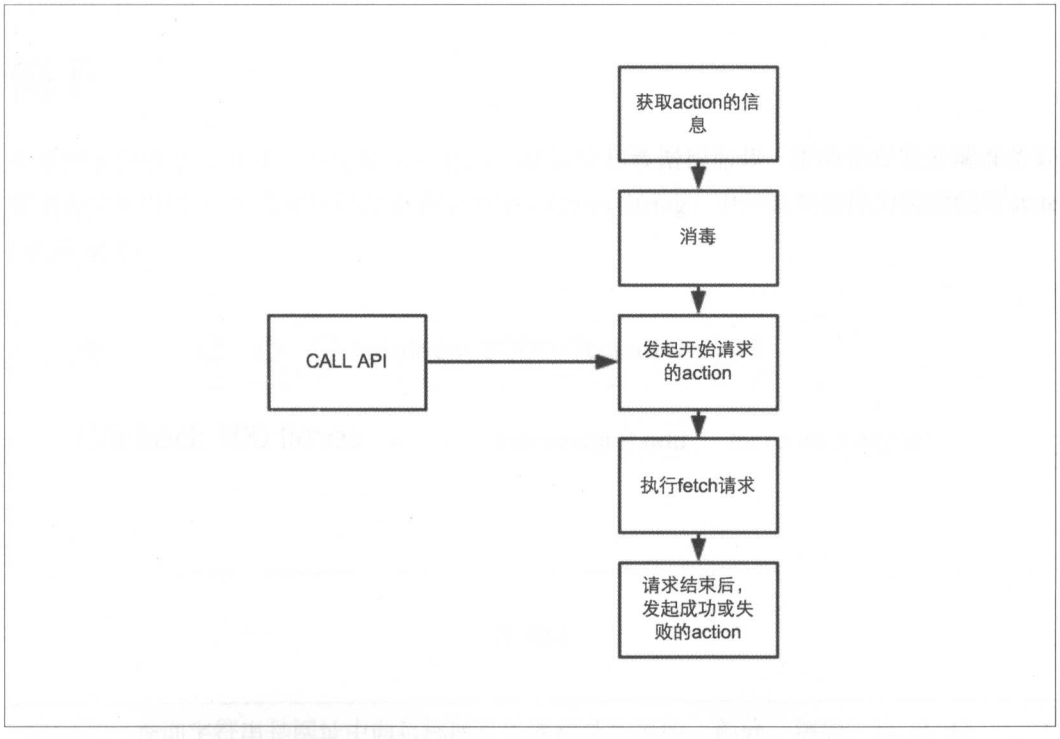


图 17-2

现在，就赶快发挥想象力编写属于自己的中间件吧！如果觉得自己写的中间件不错的话，还可以将其做成 NPM 包上传到社区，给更多的人使用。

总结

- 自定义 Redux 中间件只需要编写一个三层的嵌套函数。
- 一个异步请求通常需要编写三个 action，分别在开始请求、请求成功和请求失败时被发起。

第18章 Universal 渲染

本章将介绍如何使用 React 与 Redux 实现简单的 Universal 渲染。

例子

本章例子仍然是计数器，不过实现了 Universal 渲染和数据预加载（指的是在服务端准备好数据再渲染页面），另外还可以从查询字符串（Query String）中获取数据作为初始化的 state（见图 18-1）。

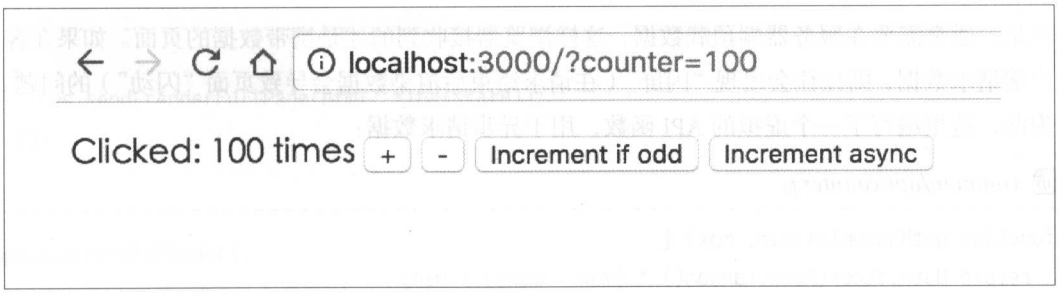


图 18-1



查询字符串是网址中向目标网页传递额外数据的一部分。例如：?p_id=42

公用一套代码

第 2 章和第 3 章的示例程序演示了同一个 React 组件分别在 Node.js 和浏览器中渲染的方法，本章将两者结合在一个程序中。不仅如此，本章的例子中要公用的也不只是 React 组件了，还增加了负责状态管理工作的 Redux。那么，这会影响程序的正常运行吗？不用担心，Redux 同样支持在多平台运行。所以不管是普通组件、容器组件，还是 Redux 的 action、reducer、中间件等都可以被公用，它们既能在服务端运行，也可以在浏览器中运行。公用的文件虽然很多，但是入口通常只有两个，一个是顶层的 React 组件，一个是 Redux 的 store。

```
import configureStore from '../common/store/configureStore';
import App from '../common/containers/App';
```

一般情况下，只要拿到它们两个，就可以运行起来整个公用的代码。

服务端渲染

实现服务端渲染非常简单，只需要使用renderToString 方法将组件渲染成字符串即可：

```
const html = renderToString(
      <Provider store={store}>
        <App />
      </Provider>
    );
```

但是，通常需要在服务器端预载数据，这样浏览器接收到的才是携带数据的页面。如果在客户端请求数据，则往往会出现“闪屏”（在请求结束后渲染数据会导致页面“闪动”）的问题。为此，这里编写了一个虚拟的 API 函数，用于异步请求数据：

common/api/counter.js

```
function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min)) + min;
}

export function fetchCounter(callback) {
  setTimeout(() => {
    callback(getRandomInt(1, 100));
  }, 500);
}
```

编写完虚拟的 API，接下来我们编写一个名为handleRender的 Express 中间件来请求 API（即预载数据），然后渲染页面。



Express 中间件是 Express 框架安置在收到请求和发出响应之间的一个处理函数。通常情况下，Express 中间件会在每次收到请求时被触发。

server/server.js

```
function handleRender(req, res) {
  fetchCounter(apiResult => {
    // qs是个npm包，用于解析查询字符串
    const params = qs.parse(req.query);
    const counter = parseInt(params.counter, 10) || apiResult || 0;

    const initialState = { counter };

    const store = configureStore(initialState);

    const html = renderToString(
      <Provider store={store}>
        <App />
      </Provider>
    );

    const finalState = store.getState();

    res.send(renderFullPage(html, finalState));
  });
}

app.use(handleRender);
```

上述代码做了这几件事：

- 使用app.use(handleRender)将中间件handleRender 连入app 服务器。
- 在中间件函数被触发时，首先执行fetchCounter()获取数据。

- 拿到数据后，并不直接将其作为初始化 state 的counter值，而是判断是否有查询字符串。如果有，则将查询字符串的数据作为counter，否则才用fetchCounter() 拿到的数据作为counter。
- 初始化 state 后，创建 store 实例。
- 然后使用Provider组件给整个程序提供 store，并将其与App 组件一起渲染成字符串。
- 最后将字符串嵌入页面发送到浏览器。

上述代码中的renderFullPage() 是自己定义的一个函数，它接受 HTML 字符串和 state 作为参数，返回一个完整的 HTML 页面：

server/server.js

```
function renderFullPage(html, initialState) {
  return `
    <!doctype html>
    <html>
      <head>
        <title>Redux Universal Example</title>
      </head>
      <body>
        <div id="app">${html}</div>
        <script>
          window.__INITIAL_STATE__ = ${JSON.stringify(initialState)}
        </script>
        <script src="/static/bundle.js"></script>
      </body>
    </html>
    `;
}
```

至此，服务端渲染以及数据预加载的流程就演示完了。

客户端渲染

服务端渲染虽然可以极大地提高页面载入速度，带来更好的 SEO，但它仅仅是吐出了一堆 HTML 字符串，静态的页面是“不会动的”。这时候，还要进行客户端渲染。

注意，刚才返回的页面中是包含打包好的 JavaScript 文件的，这就是客户端渲染的执行脚本：

```
<script src="/static/bundle.js"></script>
```

客户端渲染不需要写更多的代码，因为 Universal 渲染的意义就是使用同一套代码。所以只需要使用render() 方法将公用的组件和 state 渲染到指定 DOM 中即可：

client/index.js

```
import 'babel-polyfill';
import React from 'react';
import { render } from 'react-dom';
import { Provider } from 'react-redux';
import configureStore from '../common/store/configureStore';
import App from '../common/containers/App';

const initialState = window.__INITIAL_STATE__;
const store = configureStore(initialState);
const rootElement = document.getElementById('app');

render(
  <Provider store={store}>
    <App />
  </Provider>,
  rootElement
);
```

到此为止，整个 Universal 渲染的流程就走完了。不过，这只是一个最简单的例子，使用 Universal 渲染还需要解决更多的问题，比如如何让服务端像客户端那样使用require加载资源、如何实现 Universal 路由、如何在路由跳转时预载数据等。这些会在后续的章节中逐一讲解。

总结

- React 组件以及 Redux 容器都可以在不同平台上运行。在 Universal 渲染中，它们往往是被公用的代码。
- 实现服务端渲染非常简单，只需要使用renderToString 方法将组件渲染成字符串即可。

- 预载数据指的是在服务端准备好数据后再渲染页面，这样浏览器接收到的才是携带数据的页面。如果在客户端请求数据，则往往会出现“闪屏”的问题。
- 为了和服务端吐出的页面保持一致，客户端需要和服务端公用组件和 state。

第三部分

拓展篇

第19章

Universal 渲染神器：Webpack 同构工具

本章讲解 Webpack 同构工具（webpack-isomorphic-tools）。如今，服务端与客户端使用一套代码渲染是开发 Web 应用程序的最佳方式。既可以拥有 SEO、高速加载等这些服务端渲染的优点，同时也保留客户端渲染所带来的丰富优雅的交互效果。如果使用 Webpack 构建这样的同构程序，那么 Webpack 同构工具会成为我们的得力助手，因为它可以帮助我们在 Node.js 中使用`require()`方法调用 Node 模块以外的各种资源文件，而不会报错。

例子

本章要讲的例子一共有三个。

- 第一个是一个 Node 脚本，演示如何在服务端使用 `require()` 方法调用 PNG 格式的图片。
- 第二个是一个使用 Webpack 打包的 JavaScript 脚本，演示如何在客户端使用`require()`方法调用 PNG 格式的图片。
- 第三个是在 universal 这个示例程序的基础上添加了 Webpack 同构工具，使得服务端和客户端均可以使用`require()`方法调用图片和样式文件（见图 19-1）。

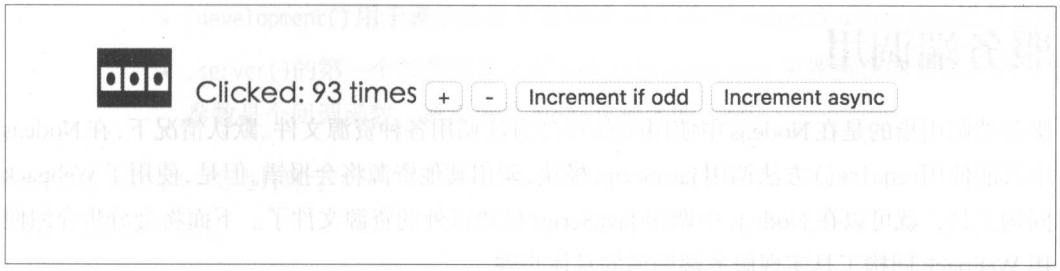


图 19-1

Webpack 同构工具的功能

使用 Webpack 可以实现在 JavaScript 中调用图片、样式、字体等资源文件，用法和调用 JavaScript 模块几乎一样：

```
require('./Counter.png')
require('./Counter.css')
```

但是，这只可以在浏览器中运行。为了实现 Universal 渲染，必须想办法让 Node.js 中的`require()`方法和浏览器环境下的`require()`方法保持一致。于是 Webpack 同构工具就诞生了。



如果你之前没有使用 Webpack 打包图片和样式文件的经验，强烈建议先去 Webpack 官网查找相关的入门例子来学习。

这个工具的工作原理是：

- 在 Webpack 打包 JavaScript 脚本时生成包含一系列键值对的 webpack-assets.json 文件。
- 给 Node.js 的 `require()` 方法打上补丁，使其可以返回 webpack-assets.json 文件中的匹配值。

从本质上来说，Universal 渲染就是服务端与客户端环境相互模拟的技术。Webpack 同构工具不依赖 React 和 Redux，只要有 Webpack，就可以使用它！

服务端调用

服务端调用指的是在 Node.js 中使用require()方法调用各种资源文件。默认情况下，在 Node.js 中只能使用require()方法调用 JavaScript 模块，调用其他资源将会报错。但是，使用了 Webpack 同构工具，就可以在 Node.js 中调用 JavaScript 模块以外的资源文件了。下面将会分步介绍使用 Webpack 同构工具实现服务端调用的具体步骤。

1. 安装这个工具。

```
npm install webpack-isomorphic-tools --save-dev
```

2. 添加 webpack-assets.json。这个文件提供了一系列的键值对，“键”对应的是require()中的参数，“值”则是require()返回值。在实际项目中，这个文件是在 Webpack 打包时候生成的，无须自己添加。

webpack-assets.json

```
{
  "assets": {
    "./Counter.png": "/dist/Counter.png"
  }
}
```

3. 编写配置指定需要加载的文件类型，然后实例化这个工具，最后在实例化对象的server()方法中填写程序的入口。

index.js

```
var WebpackIsomorphicTools = require('webpack-isomorphic-tools');

var config = { assets: { images: { extensions: ['png'] } } };

new WebpackIsomorphicTools(config)
  .development()
  .server(__dirname, function () {
    console.log(require('./Counter.png'))
  });
```

在上述代码中：

- config用于指定扩展名为png 的文件将可以被调用。

- .development()用于表示这是开发环境，将不会对 webpack-assets.json 进行缓存。
- .server()的第一个参数需要填写 webpack-assets.json 文件所在的目录，第二个参数是个回调函数，通常填写 Node.js 程序的入口。

运行这个脚本，成功打印了：

```
/dist/Counter.png
```

这表明我们成功地在 Node.js 中调用了 JavaScript 模块以外的资源文件。除了例子中的图片之外，可以更改相应的配置（即上述代码中的config），来加载字体、样式等，这些知识会在后面详细讲解。

客户端调用

客户端调用指的是在浏览器环境中使用require()方法调用各种资源文件。对于在浏览器中运行的代码而言，只要使用了 Webpack 来打包，就可以直接使用require()方法调用图片、字体、样式等多个资源。但是为了生成给服务端用的 webpack-assets.json 文件，还需要在 webpack.config.js 中加入 Webpack 同构工具这个插件。下面将分步介绍如何使用 Webpack 在客户端调用各种资源文件，并同时生成给服务端使用的 webpack-assets.json 文件。

1. 编写程序入口文件。该文件的作用是在页面中添加一张图片。

 index.js

   ```
   document.write(`<img src="${require('./Counter.png')}" alt="Counter"/>`);
   ```

2. 编写 HTML 文件，添加打包后 JavaScript 脚本。

 index.html

   ```
   <!DOCTYPE html>
   <html lang="en">
   <head>
       <meta charset="UTF-8">
       <title>Example</title>
   </head>
   <script src="dist/main.js"></script>
   <body>
   </body>
   </html>
   ```

3. 编写 Webpack 配置文件，使其既可以调用 PNG 格式的图片，又可以生成给服务端使用的 webpack-assets.json 文件。

webpack.config.js

```
var WebpackIsomorphicToolsPlugin = require('webpack-isomorphic-tools/plugin');

var config = { assets: { images: { extensions: ['png'] } } };

var webpackIsomorphicToolsPlugin = new WebpackIsomorphicToolsPlugin(config);

module.exports = {
  context: __dirname,
  entry: './index.js',
  output: {
    path: __dirname + '/dist',
    filename: 'main.js'
  },
  module: {
    loaders: [
      { test: /\.png$/, loader: 'url-loader?limit=10240' }
    ]
  },
  plugins: [webpackIsomorphicToolsPlugin]
};
```

上述代码做了这几件事：

- 设置context为项目根目录，该步骤是配置 Webapck 同构工具的必填项。
- 使用url-loader?limit=10240来加载图片文件，limit=10240表示如果文件小于 10 KB，则返回该文件的 base64 编码。
- 将webpackIsomorphicToolsPlugin这个插件加入到plugins选项。

如此一来，既可以实现 Webpack 对图片的加载，又能够生成 webpack-assets.json 给服务端使用了。

先进行打包，然后运行程序：

```
npm run build
npm start
```

结果发现，在程序目录中生成了 webpack-assets.json 文件，在浏览器中输出了一张图片。

真实场景

真实场景的例子是基于 Redux 官方示例 universal 修改得到的。下面几节将会介绍如何在真实的 Universal 应用中使用 Webpack 同构工具。

配置服务器的入口

使用 Webpack 同构工具的第一步是修改服务器的入口，使得服务器中的代码可以使用`require()`方法调用各种资源文件。

修改前：

```
require('babel-register');
require('./server');
```

修改后：

server/index.js

```
/* eslint-disable */
require('babel-register');

var path = require('path');
var rootDir = path.resolve(__dirname, '..');

var WebpackIsomorphicTools = require('webpack-isomorphic-tools');
global.webpackIsomorphicTools = new WebpackIsomorphicTools(require('../webpack-
    isomorphic-tools'))
  .development()
  .server(rootDir, function() {
    require('./server')
  });
```

上述代码所做的改动如下：

- 将服务器的入口文件放在`webpackIsomorphicTools.server()`的回调函数中。

- rootDir 则是 webpack-assets.json 文件所在的目录。
- 使用require('../webpack-isomorphic-tools') 获取同构工具的配置信息。

注意，与前面的例子相比，本例将同构工具的配置信息转移到了根目录下的 webpack-isomorphic-tools.js 中。

独立开发服务器

使用了 Webpack 同构工具后，就应该将开发服务器和发送页面的服务器分开。如果不分开，同构工具会找不到 webpack-assets.json 文件，webpackIsomorphicTools.server()中的回调函数将无法被触发，require('./server') 也不会被执行。由于开发服务器也在这个'./server' 中，所以开发服务器也不会运行，进而 Webpack 无法编译生成 webpack-assets.json 文件。如此一来，程序将一直停留在寻找 webpack-assets.json 文件的状态。

为了解决这个问题，应该将开发服务器独立出去，使其可以正常运行：

server/dev-server.js

```
/* eslint-disable */
var Express = require('express');

var webpack = require('webpack');
var webpackDevMiddleware = require('webpack-dev-middleware');
var webpackHotMiddleware = require('webpack-hot-middleware');
var webpackConfig = require('../webpack.config');

var app = new Express();
var port = 3001;

var compiler = webpack(webpackConfig);
app.use(webpackDevMiddleware(compiler, { noInfo: true, publicPath: webpackConfig.output.
    publicPath }));
app.use(webpackHotMiddleware(compiler));

app.listen(port, (error) => {
  if (error) {
    console.error(error)
  } else {
```

```
    console.info(`==> 🚧  Webpack development server listening on port ${port}.`)
  }
});
```

为了不与发送页面的服务器的端口相冲突，我们将开发服务器的端口指定为3001 。

配置 Webpack

配置完服务端程序的入口，也将开发服务器独立出来，接下来就需要配置 Webpack，使其既可以配合开发服务器完成普通的打包任务，又可以生成供服务端使用的 webpack-assets.json 文件。配置 Webpack 主要包含下面三步：

- 指定context选项为 webpack-assets.json 文件所在的目录，即根目录。
- 添加 Webpack 同构工具的插件。
- 由于本例将开发服务器独立到了3001端口上，所以还需要指定热替换的新路径为http://localhost:3001/__webpack_hmr。
- 添加style-loader和css-loader两个加载器来调用 CSS 文件。

由于指定context和添加插件两个步骤在前面已经介绍过了，这里只介绍后两个步骤。

1. 指定热替换的新路径。

 修改前：

   ```
   entry: [
       'webpack-hot-middleware/client',
       './index.js'
     ],
   ```

 修改后：

   ```
   entry: [
       'webpack-hot-middleware/client?path=http://localhost:3001/__webpack_hmr',
       './client/index.js'
     ],
   ```

2. 添加style-loader和css-loader两个加载器来调用 CSS 文件。

   ```
   { test: /\.css$/, loader: 'style-loader!css-loader?modules' }
   ```

在上述代码中：

- style-loader 用于将样式嵌入页面中。
- css-loader 用于加载样式文件。
- ?modules 表示激活 CSS Module，一旦此项被激活，CSS 中的类选择器.counter 将变为._35UgWDOiXIX_lWjbLQ3U-N 以实现 CSS 的模块化。不过你也可以通过一些参数把变化后的类选择器名称指定为更加语义化的格式，这些我们在后面会详细讲解。



CSS Module 是限制 CSS 作用域（Scope）的一种技术，它让 CSS 也拥有了类似局部作用域（Local Scope）的概念。为了实现这个功能，CSS Module 会为每个类选择器生成一个独一无二的名字，不会与其他选择器重名。

完整代码：

webpack.config.js

```
var path = require('path');
var webpack = require('webpack');

var WebpackIsomorphicToolsPlugin = require('webpack-isomorphic-tools/plugin');

module.exports = {
  context: path.resolve(__dirname),
  devtool: 'cheap-eval-source-map',
  entry: [
    'webpack-hot-middleware/client?path=http://localhost:3001/__webpack_hmr',
    './client/index.js'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: 'http://localhost:3001/static/'
  },
  plugins: [
    new webpack.optimize.OccurrenceOrderPlugin(),
    new webpack.HotModuleReplacementPlugin(),
```

```
    new WebpackIsomorphicToolsPlugin(require('./webpack-isomorphic-tools')).development
      ()
  ],
  module: {
    loaders: [
      {
        test: /\.js$/,
        loader: 'babel',
        exclude: /node_modules/,
        include: __dirname,
        query: {
          presets: ['react-hmre']
        }
      },
      { test: /\.png$/, loader: 'url-loader?limit=10240' },
      { test: /\.css$/, loader: 'style-loader!css-loader?modules' }
    ]
  }
};
```

配置 Webpack 同构工具

完成了前面几步，下面只需要配置 Webpack 同构工具就大功告成了。在前两个例子中，因为只调用了 PNG 格式的图片，所以 Webpack 同构工具的配置信息非常简单：

```
{ assets: { images: { extensions: ['png'] } } }
```

不幸的是，如果要使用style-loader和css-loader调用样式文件，配置信息就没有这么简单了。Webpack 同构工具为style-loader和css-loader单独设计了一些配置：

webpack-isomorphic-tools.js

```
var WebpackIsomorphicToolsPlugin = require('webpack-isomorphic-tools/plugin');

var config = {
    assets: {
    images: {extensions: ['png']},
    style_modules: {
      extensions: ['css'],
```

```
      filter: function(module, regex, options, log) {
        return WebpackIsomorphicToolsPlugin.style_loader_filter(module, regex, options,
            log);
      },
      path: function(module, options, log) {
        return WebpackIsomorphicToolsPlugin.style_loader_path_extractor(module, options,
            log);
      },
      parser: function(module, options, log) {
        return WebpackIsomorphicToolsPlugin.css_modules_loader_parser(module, options,
            log);
      }
    }
  }
};

module.exports = config;
```

上述代码中，style_modules 选项就是针对样式文件调用的配置。与 images 选项相比，style_modules 选项重写了filter、path和parser的默认值。filter、path和parser的默认值与针对style-loader和css-loader设计的新返回值如表 19-1 所示。

表 19-1

	默认值	针对 style-loader 和 css-loader 设计的新返回值
filter	regex.test(module.name)	WebpackIsomorphicToolsPlugin.style_loader_filter(module, regex, options, log)
path	module.name	WebpackIsomorphicToolsPlugin.style_loader_path_extractor(module, options, log)
parser	module.source	WebpackIsomorphicToolsPlugin.css_loader_parser(module, options, log) WebpackIsomorphicToolsPlugin.css_modules_loader_parser(module, options, log)

如果你想知道上述配置和默认配置有什么区别，可以将其打印出来或者直接查看源代码。

如果你想了解更详细的 Webpack 同构工具的用法，请查阅官网文档（https://github.com/halt-hammerzeit/webpack-isomorphic-tools）。

至此，就可以在公用的代码中使用require()方法调用图片和样式文件了。

在服务端和客户端使用 require()

下面就是在服务端和客户端使用require()调用图片的示例代码：

common/components/Counter.js

```
function Counter({ increment, incrementIfOdd, incrementAsync, decrement, counter }) {
  return (
    <p>
      <img
        alt="Counter"
        src={require('./Counter.png')}
        className={require('./Counter.css').counter}
      />
      Clicked: {counter} times
      {' '}
      <button onClick={increment}>+</button>
      {' '}
      <button onClick={decrement}>-</button>
      {' '}
      <button onClick={incrementIfOdd}>Increment if odd</button>
      {' '}
      <button onClick={() => incrementAsync()}>Increment async</button>
    </p>
  );
}
```

上述代码中，因为使用了 CSS Module，所以应该使用require('./Counter.css').counter 作为类名。require('./Counter.css').counter 将会返回 CSS Module 生成的类名。

运行程序，将会发现图片有一个由大变小的过程，这是因为服务端吐出的页面没有样式，需要等到客户端使用style-loader将样式写入页面后才会生效。所幸这是开发环境，如果在生产环境下，应该把 CSS 文件提取出来，作为单独的文件嵌入页面上方，这样就不会出现上述的“闪屏”问题了。关于如何将 CSS 文件提取出来这个问题，我们会在后续章节详细讲解。

Webpack 同构工具的使用并不简单，甚至 Webpack 这个工具本身的使用就非常复杂。这些构建工具所带来的高昂的学习成本让 JavaScript 界进入到了前所未有的“疲劳期”。有时我们会想，如果使用传统方式来加载图片和样式文件等资源，不就不用如此辛苦地进行各种设置了吗？但要是真那么做的话，我们也无法享受前端自动化构建带来的各种便利了。总之，前端工程中还有太多需要提升的地方，这对开发者而言，既是痛苦，也是机遇。如果你能挺身而出解决这些问题，那么下一个前端明星就是你！

总结

- 从本质上来说，Universal 渲染就是服务端与客户端环境相互模拟的技术。
- Webpack 同构工具的工作原理是更改 Node.js 的`require()`方法，使其拥有与客户端一样的功能。

第20章

多页面的实现：路由

本章将介绍 React Router 的相关知识，最后搭建一个支持 Universal 渲染的路由系统。

例子

本章例子拥有两个页面，一个是主页，一个是计数器页面。当进入指定页面时，顶部的导航栏将被激活，变为红色（见图 20-1，本书为黑白印刷，此图显示为灰度图，只能看出灰度区别）。

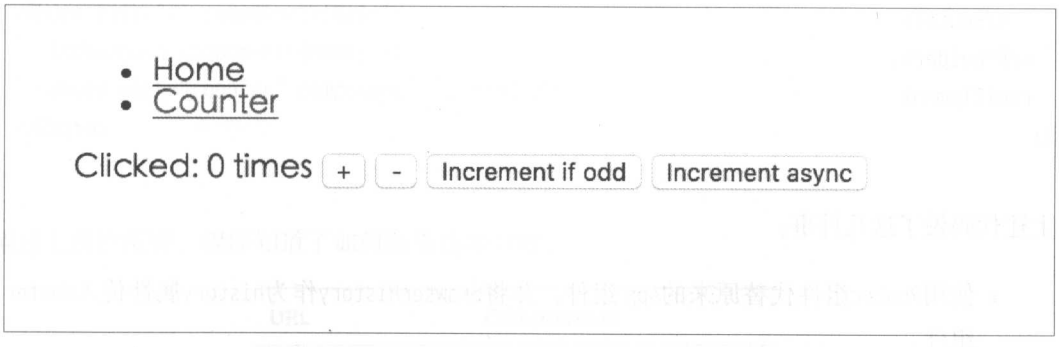


图 20-1

路由也是组件

在 React Router 中，路由的本质不过是嵌套的视图组件，所以使用路由就像是使用其他组件那样，直接将其渲染出来即可。

client/index.js

```
import 'babel-polyfill';
import React from 'react';
import { render } from 'react-dom';
import { Provider } from 'react-redux';
import { Router, browserHistory } from 'react-router';

import configureStore from '../common/store/configureStore';
import getRoutes from '../common/routes';

const initialState = window.__INITIAL_STATE__;
const store = configureStore(initialState);
const rootElement = document.getElementById('app');
const routes = getRoutes();
const history = browserHistory;

render(
  <Provider store={store}>
    <Router history={history}>
      {routes}
    </Router>
  </Provider>,
  rootElement
);
```

上述代码做了这几件事：

- 使用Router组件代替原来的App 组件，并将browserHistory作为history属性传入Router组件。
- 使用getRoutes() 获取路由匹配信息，并将其作为Router组件的子元素渲染出来。关于路由匹配，会在下一节中讲解。

与Router组件一样，browserHistory也来自 React Router，它的功能是监听浏览器地址栏的变化，解析 URL 并与路由组件进行匹配。如此一来，React Router 才可以在不同的 URL 下渲染出正确匹配的组件。除此之外，browserHistory还提供了一些方法可以让你在代码中进行导航：

```
import { browserHistory } from 'react-router'

// 跳转到某个路径
browserHistory.push('/some/path')

// 回退到先前的地址
browserHistory.goBack()
```

路由匹配

仅仅渲染路由组件是不够的，还需要配置路由匹配信息，告诉路由如何根据 URL 来显示相应的子组件。

common/routes.js

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';
import Counter from './containers/Counter';
import Main from './containers/Main';
import Home from './containers/Home';

export default () => (
  <Route path="/" component={Main}>
    <IndexRoute component={Home} />
    <Route path="counter" component={Counter} />
  </Route>
);
```

通过上面的配置，程序知道了如何渲染这些 URL：

URL	Components
/	Main -> Home
/counter	Main -> Counter

这里使用了嵌套路由的概念，所以才可以显示嵌套的多重视图：Main -> Counter 。另外，上述代码中的IndexRoute组件将作为父级Route的默认子路由组件，也就是 URL 为/时显示的组件。

完成了路由组件的渲染和路由匹配这两步，下面我们使用 React Router 提供的Link和IndexLink组件为用户提供导航的操作。

使用 Link 和 IndexLink 导航

Link组件的功能和 HTML 中的<a/>标签相似，但是除了普通的导航功能外，它还拥有其他一些实用的属性，比如本例使用了activeStyle属性来让Link组件在激活状态下变为红色。那么什么时候Link组件会处于激活状态呢？比如说，如果你使用了<Link to="/">Home</Link> 去导航/这个 URL，那么只要进入以/开头的 URL 时，Link组件都会被激活。不过，如此一来，岂不是所有页面下的<Link to="/">Home</Link>都处于激活状态了吗？这显然不是我们想要的效果，于是 React Router 又提供了IndexLink这个组件去导航类似/的 URL，来避免Link组件的副作用。下面就是Link组件和IndexLink组件的简单用法示例。

common/containers/Main.js

```
import React, { PropTypes } from 'react';
import { Link, IndexLink } from 'react-router';

function Main(props) {
  return (
    <div>
      <ul>
        <li><IndexLink to="/" activeStyle={{ color: 'red' }}>Home</IndexLink></li>
        <li><Link to="/counter" activeStyle={{ color: 'red' }}>Counter</Link></li>
      </ul>
      {/* this will render the child routes */}
      {React.cloneElement(props.children, props)}
    </div>
  );
}
Main.propTypes = {
  children: PropTypes.any.isRequired
```

```
};

export default Main;
```

至此，一个可以在客户端运行的路由系统就完成了。

服务端路由

如果想要实现真正的多页面程序，仅仅依靠客户端路由是不行的，因为我们看到的页面仍然只有 JavaScript 脚本，没有数据。这不仅会影响 SEO，而且因为浏览器必须等到脚本加载完成才能渲染页面，所以还会导致页面首次加载速度很慢，而影响用户体验。为了解决这个问题，我们必须实现服务端路由，即让“分页”这个工作在服务端就完成好，服务端吐出的直接就是携带模板和数据的页面，不需要等到 JavaScript 脚本来渲染。值得高兴的是，React Router 就支持服务端路由。



服务端路由也需要加载 JavaScript 文件，但是因为 HTML 的解析是由上而下进行的，所以等到加载页面底部的 JavaScript 文件时，上面的模板以及样式已经显示出来了。

在 React Router 中实现服务端路由非常简单，只需要在 Express 中间件（不限制只能 Express 中间件）中加上一个用于匹配路由的`match`函数，并在`match`的回调中进行渲染就可以了。

server/server.js

```
function handleRender(req, res) {
  const initialState = { counter: 0 };
  const store = configureStore(initialState);
  const routes = getRoutes();

  match({ routes, location: req.url }, (err, redirect, renderProps) => {
    const html = renderToString(
        <Provider store={store}>
          <RouterContext {...renderProps} />
        </Provider>
      );
      const finalState = store.getState();
```

```
    res.send(renderFullPage(html, finalState));
  });
}
```

在上述代码中，match 用来匹配 URL 和要渲染的路由组件。它的第一个参数是一个对象，内容包括所有路由的集合和location（即当前请求的 URL）。它的第二个参数是一个回调函数，渲染工作将被转移到这个回调函数中。RouterContext这个组件则用于接受renderProps来渲染路由组件。

为何需要使用match这个函数呢？这是因为match这个函数是专为异步路由设计的，而服务端的路由本身又是异步的。假如客户端路由有异步操作，同样应该使用match进行匹配，不过本例中不需要这样做。关于异步路由的相关知识，我们会在后续章节中详细讲解。

上述代码只是服务端路由的一个基本用法示例。在真实项目中往往还需要增加处理重定向、500 和 404 错误的功能。这些功能实现起来也非常简单，只需要先从match的回调函数中拿到err、redirect、renderProps三个参数，然后依据这些参数来做出不同的响应即可：

```
function handleRender(req, res) {
  const initialState = { counter: 0 };
  const store = configureStore(initialState);
  const routes = getRoutes();

  match({ routes, location: req.url }, (err, redirect, renderProps) => {
    if (redirect) {
      res.redirect(redirect.pathname + redirect.search);
    } else if (err) {
      console.error('ROUTER ERROR:', err.stack);
      res.status(500);
    } else if (renderProps) {
      res.status(200);
      const html = renderToString(
        <Provider store={store}>
          <RouterContext {...renderProps} />
        </Provider>
      );
      const finalState = store.getState();
      res.send(renderFullPage(html, finalState));
    } else {
```

```
      res.status(404).send('Not found');
    }
  });
}
```

match会根据不同的请求，在回调函数中返回不同的参数。常见情况下的参数如下：

- 在正确匹配，没有重定向的情况下，前两个参数都是null，renderProps包含了匹配的组件和location等信息。
- 在有重定向的情况下，redirect将包含重定向信息。
- 在出现内部错误的情况下，err将包含错误信息。
- 在 404 错误的情况下，三个参数均为undefined。

到此为止，整个支持 Universal 渲染的路由系统就搭建完成了。当然，React Router 的用法远不止这些，本章仅仅介绍了最常用的几个功能。在后续章节中，我们还会继续介绍 React Router 的相关知识。与此同时，你也应该配合 React Router 的官网文档（https://github.com/reactjs/react-router）进行学习。

总结

- 路由本质上只是一个多重视图的组件。
- 配置路由匹配信息，可以告诉路由如何根据 URL 来运行和显示相应的组件。
- Link组件的功能和<a/>标签相似，但是它支持一些可用于激活状态的属性。IndexLink组件则用来避免Link组件导航／时产生的副作用。
- 要实现服务端路由，只需要在 Express 中间件中加上一个匹配路由的函数，并在其回调中进行渲染即可。

第21章

多页面下的异步操作

本章首先会介绍一个实用的异步工具——redux-amrc，然后使用 redux-amrc 在 React 与 Redux 搭建的多页面程序中进行异步操作。

例子

本章例子是在路由示例的基础上改写的，它所实现的功能包括：

- 在计数器页面渲染之前预载数据。
- 通过单击“+”/“-”按钮操作从服务器传来的数据。
- 通过单击“load”按钮刷新数据
- 显示程序当前的 state（见图 21-1）。

redux-amrc

在进行异步操作前，我们先来学习一个实用的异步工具——redux-amrc。该工具和“多页面”没有关系，也不依赖 React，只要你的 Redux 程序中需要处理异步，就可以使用它！

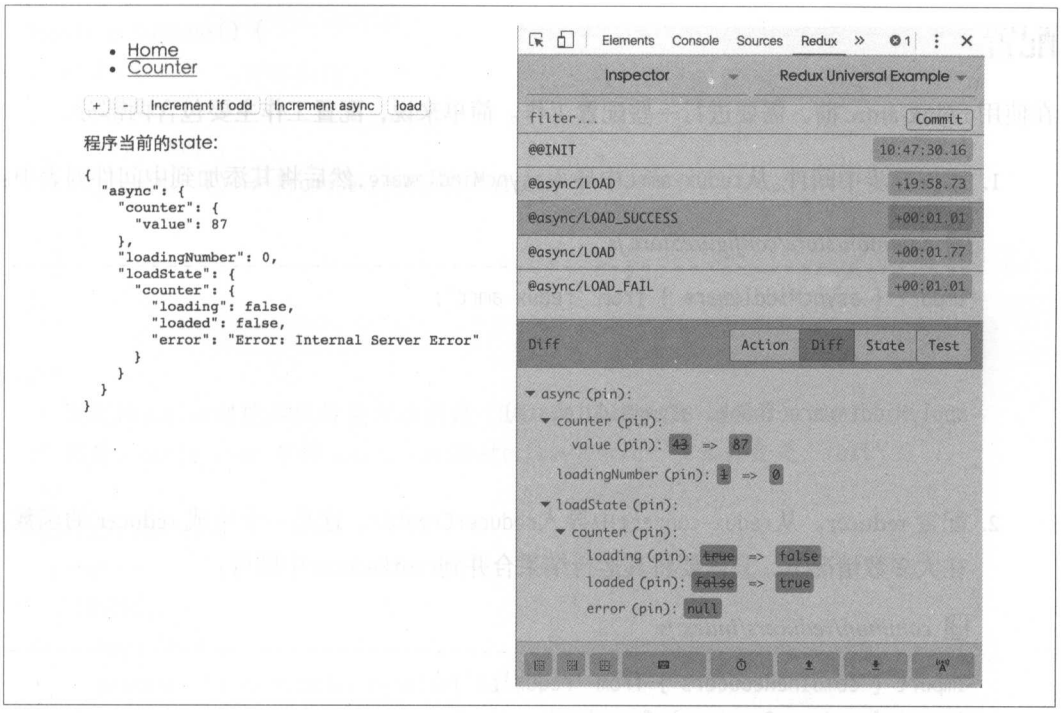


图 21-1



redux-amrc 是 Redux async middleware and reducer creator 的缩写，即 Redux 异步中间件和 reducer 创建函数。

功能

在 React 与 Redux 的应用中，我们一般是这样从服务器请求数据的：

- 编写 action 创建函数，执行网络请求加载数据。
- 判断网络请求是否成功，并在网络请求前后分别发起相应的 action。
- 编写 reducer，根据相应的 action 来更新 state。

这个过程非常烦琐，我们期望能够用最少的代码完成所有步骤。redux-amrc 的功能就是将以上过程封装起来。我们只需要将网络请求写进 Promise 传给 redux-amrc 就可以了，redux-amrc 会完成接下来所有的工作。下面几节将详细介绍 redux-amrc 的配置和使用方法。

配置

在使用 redux-amrc 前，需要进行一些配置工作。简单来说，配置工作主要包含两部分。

1. 添加异步中间件。从redux-amrc中导入asyncMiddleware，然后将其添加到中间件列表中：

 common/store/configureStore.js

   ```
   import { asyncMiddleware } from 'redux-amrc';

   ...
   applyMiddleware(thunk, asyncMiddleware)
   ...
   ```

2. 配置 reducer。从redux-connect中导入reducerCreator，这是一个生成 reducer 的函数，在大多数情况下，只需要将其运行结果合并到rootReducer中即可：

 common/reducers/index.js

   ```
   import { combineReducers } from 'redux';
   import { reducerCreator } from 'redux-amrc';

   const rootReducer = combineReducers({
     async: reducerCreator()
   });

   export default rootReducer;
   ```

至此，配置工作就完成了。

使用

redux-amrc的使用非常简单，只需要编写特定的 action 创建函数并发起即可：

```
import { ASYNC } from 'redux-amrc';

/**
 * 发起该action创建函数将会自动创建 LOAD 和 LOAD_SUCCESS 这两个action,
 * 同时，state tree 中的 async.[key] 节点将会是 'success'
 */
```

```
function success() {
  return {
    [ASYNC]: {
      key: 'key',
      promise: () => Promise.resolve('success')
    }
  }
}

/**
 * 发起该action创建函数将会自动创建 LOAD 和 LOAD_FAIL 这两个action,
 * 同时, state tree 中的 async.loadState.[key].error 节点将会是 'fail'
 */
function fail() {
  return {
    [ASYNC]: {
      key: 'key',
      promise: () => Promise.resolve('fail')
    }
  }
}
```

当发起上述 action 创建函数后，redux-amrc 将会自动地在 Promise 被解决（或被拒绝）的前后发起相应的 action，并通过 redux-amrc 内置的 reducer 更新相应的 state。这些 action 和 reducer，无须开发者自己编写。

Action 与 state

redux-amrc 用到如下的 action 和 state。

- action
 - LOAD：数据开始加载。
 - LOAD_SUCCESS：数据加载成功。
 - LOAD_FAIL：数据加载失败。
- state
 - async.[key]：加载成功后的数据。

- async.loadState.[key].loading：数据正在加载。
- async.loadState.[key].loaded：数据加载完成。
- async.loadState.[key].error：数据加载失败的理由。
- async.loadingNumber：正在加载的数量。

在开发过程中，可以通过 Redux 开发者工具查看这些 action 和 state（见图 21-2）。

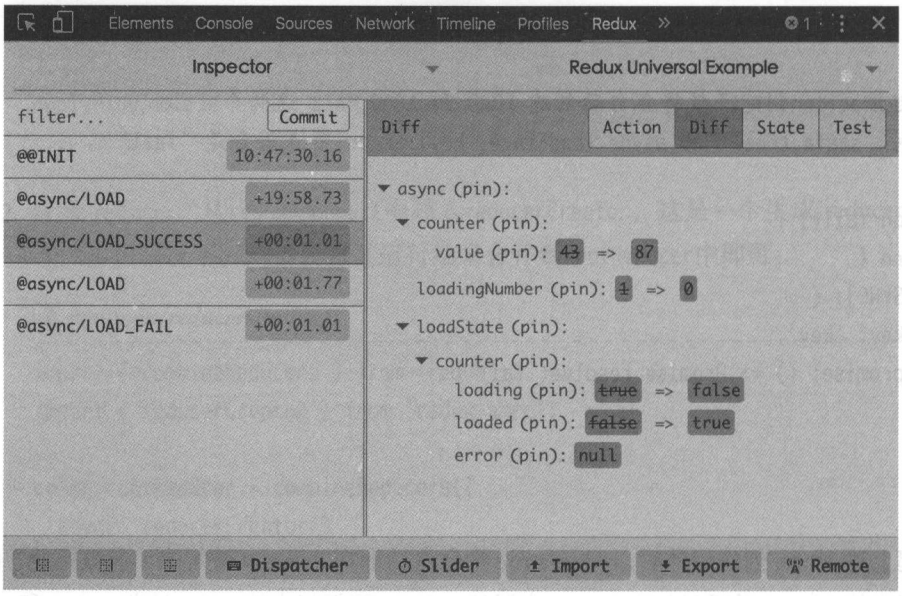


图 21-2

API

redux-amrc 的接口有三个，分别是asyncMiddleware、ASYNC和reducerCreator，它们的含义及用法如下：

- asyncMiddleware对应 redux-amrc 中的“am”，它是一个异步中间件，用于封装一些常用的异步逻辑。
- ASYNC是 redux-amrc 提供的一个 Symbol 对象，用于让异步中间件从众多 action 创建函数中筛选出指定的 action 创建函数。它被作为对象的属性由 action 创建函数返回。该属性也接受一个对象，这个对象包含key和promise两个属性。

 - key属性接受一个字符串。
 - promise属性则接受一个函数。该函数的返回值是一个 Promise 对象。如果 Promise 对象被解决，那么已解决的数据最后会传递给 Redux 的state.async.[key]。如果 Promise 被拒绝，那么被拒绝的理由将会传递给 Redux 的state.async.loadState.[key].error。

- reducerCreator 对应 redux-amrc 中的“rc”，它是一个 reducer 创建函数，即生成 reducer 的函数。大多数情况下，只需要直接运行 reducerCreator，并将其运行结果合并到 rootReducer中即可，但该函数也可以接受 reducers 作为可选参数，以便更新 state.async.[key] 节点上的数据。
 - reducers 是由若干个 key 和 reducer 所组成的键值对，“键”对应 ASYNC 中的 key，“值”是更新 state.async.[key] 的 reducer。其实，这里的 reducers 与 combineReducers(reducers) 中的 reducers 格式是一样的。

有了 redux-amrc，我们再也不用编写大量的样板代码了，只需要将 Promise 对象和指定的key值传给 redux-amrc 即可。下面我们将在多页面程序中使用 redux-amrc 来进行一些异步操作。

多页面下的异步操作

在一个多页面的 Web 程序中，我们通常会做三件与异步相关的事情：

- 在页面加载前，从服务器请求数据，然后渲染页面。
- 在页面加载后，“手动地”与服务器通信，比如通过单击按钮刷新数据或发送一个 POST 请求等。
- 在页面加载后，操作从服务器传来的数据，比如增/删/改/查等。

下面将会演示如何使用 redux-amrc 完成上述三个场景中的操作。

编写测试 API

在演示之前，需要先编写一个用于测试的 API。为了能看到网络请求中的异常情况，应该让这个 API 有 33%的概率会抛出 500 错误。

common/api/counter.js

```
module.exports = function(app) {
  app.get('/api/counter', (req, res) => {
    setTimeout(function () {
      if (Math.random() < 0.33) {
        res.status(500).end();
      } else {
        res.json({
          value: getRandomInt(1, 100)
        });
      }
    }, 1000)
  })
};

function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min)) + min;
}
```

上述代码首先使用setTimeout模拟请求中的延迟情况，然后通过Math.random() < 0.33实现在指定概率下抛出异常，最后在成功响应中返回 1 到 100 的随机数。

预载数据

编写完了用于测试的 API，下面来演示如何预载数据。使用 redux-amrc 预载数据非常简单，只需要两步即可完成：

1. 编写 redux-amrc 定制的 action 创建函数。

 actions/index.js

   ```
   import { ASYNC } from 'redux-amrc';

   export function load() {
     return {
       [ASYNC]: {
         key: 'counter',
         promise: () => fetch('http://localhost:3000/api/counter')
   ```

```
        .then(res => {
          if (!res.ok) {
            throw new Error(res.statusText);
          }
          return res.json();
        })
    }
  };
}
```

本例中，传给该 action 创建函数的 Promise 对象是fetch方法返回的。这里首先使用了fetch方法请求 API，然后在then()方法中判断请求是否成功，成功则返回res.json()，失败则抛出异常。属性key则指定为counter。

2. 在路由组件的onEnter属性中编写方法发起上述 action 创建函数。

common/routes.js

```
import { load } from './actions';

export default store => {
  const preload = (nextState, replace, cb) => {
    if (__SERVER__ || nextState.location.action === 'PUSH') {
      store.dispatch(load()).then(() => cb());
    } else {
      cb();
    }
  };
  return (
    <Route path="/" component={Main}>
      <IndexRoute component={Home}/>
      <Route path="counter" component={Counter} onEnter={preload}/>
    </Route>
  );
};
```

在上述代码中，onEnter是路由组件的一个属性。onEnter所接受的函数会在即将进入路由之前被触发。同时，它还提供了nextState和replace两个参数作为下一个路由状态和重定向函数。如果列出了第三个参数cb，onEnter所接受的函数将会异步运行，路由跳转也将被阻塞，直到cb被调用。

条件判断__SERVER__ || nextState.location.action === 'PUSH'的作用则是限制只在服务端和客户端跳转时进行数据预载，这样做可以防止服务端渲染完成后客户端重复预载数据。这段代码包含了两个参数的用法，其解释如下。

- __SERVER__：该参数是个全局变量，它用于判断当前代码是否运行于服务端。该参数的定义是在服务端入口以及 Webpack 插件中进行的。这里不再列出代码清单，可自行查看。
- nextState.location.action：该参数来自 React Router，它描述了 URL 变化的类型。PUSH代表从现有页面跳转到其他页面。其他类型请参考 React Router 的官网文档。

注意，由于onEnter中包含异步操作，此时应该使用match来匹配渲染客户端路由，match是专门为异步路由所设计的方法，它的具体用法已经在前面的章节中介绍过了，这里不再赘述。客户端路由的代码如下：

```
...

match({ history, routes }, (err, redirect, renderProps) => {
  if (redirect) {
    history.replace(redirect);
  } else if (err) {
    history.goBack();
    console.error(err.stack);
  } else {
    render(
      <Provider store={store}>
        <Router {...renderProps} />
      </Provider>,
      rootElement
    );
  }
});
```

至此，预载数据的操作就完成了。简单来说，预载数据是在onEnter中发起包含网络请求的 action 创建函数。运行程序，在浏览器中打开计数器页面，会发现页面在载入前延迟了 1 秒，而且服务器吐出的已经是包含数据的页面了。

手动加载数据

与预载数据相比，手动加载数据只不过是换了 action 创建函数的发起位置而已。预载数据是在路由的onEnter中进行的，而手动加载数据则是在用户操作所触发的函数中进行的。

使用 react-redux 将上一节中的 action 创建函数连接到组件上，并在单击“load”按钮时发起它：

common/containers/Counter.js

```
import React, { Component, PropTypes } from 'react';
import { connect } from 'react-redux';
import * as actionCreators from '../actions';

@connect(
  state => ({ state }),
  actionCreators
)
class Counter extends Component { // eslint-disable-line

  static propTypes = {
    increment: PropTypes.func.isRequired,
    incrementIfOdd: PropTypes.func.isRequired,
    incrementAsync: PropTypes.func.isRequired,
    decrement: PropTypes.func.isRequired,
    load: PropTypes.func.isRequired,
    state: PropTypes.object.isRequired
  };

  render() {
    const {
      increment, decrement, incrementIfOdd, incrementAsync, state, load
    } = this.props;
    return (
      <div>
        <button onClick={increment}>+</button>
        {' '}
        <button onClick={decrement}>-</button>
        {' '}
        <button onClick={incrementIfOdd}>Increment if odd</button>
```

```
        {' '}
        <button onClick={() => incrementAsync()}>Increment async</button>
        {' '}
        <button onClick={load}>load</button>
        <br/><br/>
        程序当前的state:
        <pre>{JSON.stringify(state, null, 2)}</pre>
      </div>
    );
  }
}

export default Counter;
```

至此，我们就可以在页面渲染后“手动地”发起网络请求了。单击程序中的“load”按钮，将会看到计数器的当前数值被来自服务器的新数据所替代，如果请求失败，错误信息将会显示在 state tree 的async.loadState.counter.error节点上。

操作数据

对于大多数展示性页面而言，预载数据和手动加载数据的功能已经够用了。但是，有时候也需要对从服务器传来的数据进行各种操作。在服务器正确响应的情况下，数据都被托管在state.async.[key]节点上。想要更新state.async.[key]节点上的数据，应该将key和更新key的 reducer 作为键值对传入reducerCreator中：

common/reducers/index.js

```
import { combineReducers } from 'redux';
import { reducerCreator } from 'redux-amrc';
import counter from './counter';

const rootReducer = combineReducers({
  async: reducerCreator({ counter })
});

export default rootReducer;
```

在上述代码中，{ counter } 是键值对 { counter: counter } 的简写。前面的 counter 键对应 ASYNC 中的 key: 'counter'。而后面的 counter 则是处理 INCREMENT_COUNTER 和 DECREMENT_COUNTER 的 reducer，它的代码与前面章节中的相同，所以这里不再列出代码清单。

如此一来，当发起INCREMENT_COUNTER和DECREMENT_COUNTER，就能够操作 redux-amrc（即state.async.counter）中的数据了。

总结

- redux-amrc 封装了 Redux 中的重复性异步操作。只需要将 Promise 和key值传给 redux-amrc，它会完成接下来所有的异步工作。
- 在路由组件的onEnter中发起 redux-amrc 定制的 action 创建函数，可以实现数据预载。
- 在用户操作所触发的函数中发起 redux-amrc 定制的 action 创建函数，可以实现手动加载数据。
- 想要操作 redux-amrc 中的数据，应该将处理 action 的 reducer 组合为对象，然后将该对象作为参数传入reducerCreator中。

第22章

使用 Bootstrap

本章介绍如何使用 Webpack 加载 Bootstrap，所用的技术包括 bootstrap-loader、bootstrap-sass、postcss-loader、Autoprefixer、React-Bootstrap 等。

例子

本章例子不仅使用了 Webpack 加载 Bootstrap，而且还自定义了一部分主题变量（可以看到部分色调与默认的不同），最后从 React-Bootstrap 中挑选了一些常用的组件进行渲染（见图 22-1）。

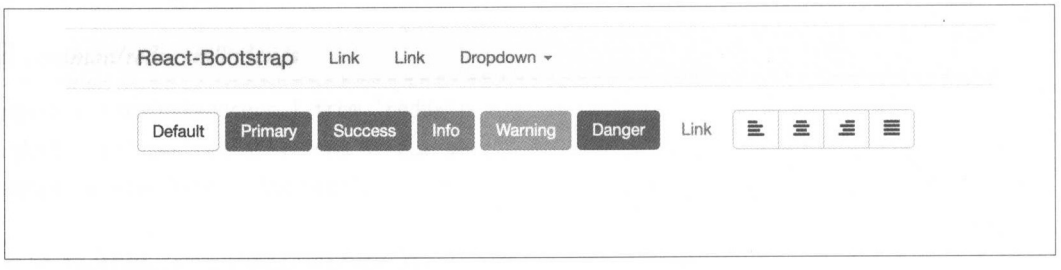


图 22-1

运行方法：

```
npm install
npm start
```

如果你的网络状况不佳，无法下载安装node-sass，请使用这个命令进行安装：

```
SASS_BINARY_SITE=https://npm.taobao.org/mirrors/node-sass/ npm install
```

bootstrap-loader

bootstrap-loader 是一个用来加载 Bootstrap 的 Webpack 加载器。这个加载器使用 Sass 处理 CSS 样式，对于 Bootstrap 3&4 都能支持。本章的示例程序使用了 Bootstrap 3。下面将介绍使用 bootstrap-loader 的具体细节。

基本用法

首先，需要安装 bootstrap-loader 及其相关依赖包：

```
npm install bootstrap-loader bootstrap-sass css-loader node-sass resolve-url-loader sass
    -loader style-loader url-loader --save-dev
```

然后在 Webpack 配置文件入口添加bootstrap-loader：

```
entry: [
    'webpack-hot-middleware/client',
    'bootstrap-loader',
    './index.js'
  ],
```

最后配置样式和字体的加载器：

```
{
  test: /\.css$/,
  loaders: [
    'style',
    'css'
  ]
},
{
  test: /\.scss$/,
  loaders: [
    'style',
    'css',
```

```
    'sass'
  ]
},
{
  test: /\.woff2?(\?v=[0-9]\.[0-9]\.[0-9])?$/,
  loader: "url?limit=10000"
},
{
  test: /\.(ttf|eot|svg)(\?[\s\S]+)?$/,
  loader: 'file'
}
```

至此，就可以在程序中使用 Bootstrap 了。不过这里一般需要对 Bootstrap 进行一些配置，尽管这些配置是可选的。

配置

配置 Bootstrap 只需要在项目根目录下添加.bootstraprc文件即可，.bootstraprc支持YAML或者JSON两种写法。

```
---
# Bootstrap版本
bootstrapVersion: 3

# Bootstrap 3 中自定义图标字体路径的开关
useCustomIconFontPath: false

# Webpack 加载器列表，注意顺序
styleLoaders:
  - style
  - css
  - sass

# 导出样式到单独的CSS文件
# 可以根据 NODE_ENV 这个环境变量设置此项
# 也可以在Webpack中配置
#   entry: 'bootstrap-loader/extractStyles'
extractStyles: false
```

```
# env:
#   development:
#     extractStyles: false
#   production:
#     extractStyles: true

# 自定义Bootstrap 变量，在Bootstrap加载前加载，用于重写默认变量，相当于自定义
    Bootstrap
preBootstrapCustomizations: ./theme/bootstrap/pre-customizations.scss

# 自定义Bootstrap 变量，在Bootstrap加载后加载，用于重写Bootstrap变量，相当于重写
    Bootstrap
bootstrapCustomizations: ./theme/bootstrap/customizations.scss

# 在此处导入自定义样式
# appStyles: ./app/styles/app.scss

### Bootstrap 样式
styles:

  # Mixins
  mixins: true

  # Reset and dependencies
  normalize: true
  print: true
  glyphicons: true

  # Core CSS
  scaffolding: true
  type: true
  code: true
  grid: true
  tables: true
  forms: true
  buttons: true

  # Components
```

```
  component-animations: true
  dropdowns: true
  button-groups: true
  input-groups: true
  navs: true
  navbar: true
  breadcrumbs: true
  pagination: true
  pager: true
  labels: true
  badges: true
  jumbotron: true
  thumbnails: true
  alerts: true
  progress-bars: true
  media: true
  list-group: true
  panels: true
  wells: true
  responsive-embed: true
  close: true

  # Components w/ JavaScript
  modals: true
  tooltip: true
  popovers: true
  carousel: true

  # Utility classes
  utilities: true
  responsive-utilities: true

### Bootstrap scripts
scripts: false
```

上述配置做了这几件事：

- 指定 Bootstrap 版本为 Bootstrap 3。

- 关闭自定义图标字体。
- 指定样式加载器。
- 设置不导出单独的 CSS 文件（如在生产环境，将设置为导出）。
- 自定义了 Bootstrap 变量，分为preBootstrapCustomizations、bootstrapCustomizations两种类型。
- 开启部分样式，关闭所有的脚本，以减小静态文件大小。

在上述配置中，preBootstrapCustomizations、bootstrapCustomizations的功能非常接近，但并不相同。

- preBootstrapCustomizations中定义的变量在原始 Bootstrap 变量导入前加载。
- bootstrapCustomizations中定义的变量则在 Bootstrap 变量导入后加载。

在讨论变量加载顺序会带来什么的不同之前，需要先了解一个知识点，即在 Sass 中：

- 重写变量是写在变量后面的。
- 重写默认变量是写在默认变量前面的。

我们来看几个示例。

无!default：

```
$color:red;
$color:blue;
p{
    color:$color;//blue
}
```

有!default：

```
$color:red;
$color:blue !default;
p{
    color:$color;//red
}
```

Bootstrap 写了一套默认变量（http://getbootstrap.com/customize/），如果你想重写这些默认变量，那么就在preBootstrapCustomizations中编写吧：

src/theme/bootstrap/pre-customizations.scss

```
$brand-primary: #5cb85c;
```

如果你想通过引用 Bootstrap 变量来重写一些变量，那么就在bootstrapCustomizations编写吧：

src/theme/bootstrap/customizations.scss

```
$btn-primary-bg: adjust-hue($btn-primary-bg, +70deg);
$btn-primary-border: $btn-primary-bg;
```

PostCSS 与 Autoprefixer

简介

PostCSS 是一个使用 JS 插件转换样式的工具。这个工具拥有丰富的功能，包括：对 CSS 代码进行语法检查、支持变量和 mixins、编译未来的 CSS 语法等等。PostCSS 被很多巨头所使用，包括 Wikipedia、Twitter、Alibaba 和 JetBrains 等。Autoprefixer 是最流行的 PostCSS 插件之一，它可以解析 CSS，并给 CSS 规则添加前缀。

编写没有前缀的 CSS：

```
:fullscreen a {
    display: flex
}
```

Autoprefixer 将帮你把它转换为：

```
:-webkit-full-screen a {
    display: -webkit-box;
    display: flex
}
:-moz-full-screen a {
    display: flex
}
:-ms-fullscreen a {
    display: -ms-flexbox;
    display: flex
}
```

```
:fullscreen a {
    display: -webkit-box;
    display: -ms-flexbox;
    display: flex
}
```

用法

本章的示例程序使用了 postcss-loader 来在 Webpack 中调用 PostCSS 插件，postcss-loader 的用法非常简单，只需两步就能将其添加到 Webpack 中。

1. 在样式文件的加载器列表中添加postcss：

```
{
  test: /\.css$/,
  loaders: [
    'style',
    'css',
    'postcss'
  ]
},
{
  test: /\.scss$/,
  loaders: [
    'style',
    'css',
    'postcss',
    'sass'
  ]
}
```

不幸的是，postcss-loader 不能直接和 CSS Module 公用。因此，如果你使用了 CSS Module，应该添加 importLoaders 参数来使它们正常工作。importLoaders 的值是 css-loader 后面的加载器的个数。

```
{
  test: /\.css$/,
  loaders: [
    'style',
```

```
        'css?modules&importLoaders=1&localIdentName=[name]__[local]__[hash:base64
            :5]',
        'postcss'
      ]
    },
    {
      test: /\.scss$/,
      loaders: [
        'style',
        'css?modules&importLoaders=2&localIdentName=[name]__[local]__[hash:base64
            :5]',
        'postcss',
        'sass'
      ]
    }
```

另外，这里还设置了localIdentName参数来指定 CSS Module 中类选择器的命名格式。

2. 在 Webpack 配置文件中增加postcss选项，并在该选项中添加插件列表。因为本例只使用了 Autoprefixer 这一个插件，故列表中只有一个autoprefixer。

```
postcss: [autoprefixer({ browsers: ['last 2 versions'] })]
```

Autoprefixer 使用了 Browserslist，所以你可以使用last 2 versions或 > 5%这样的查询参数来指定项目将要运行的浏览器对象。Browserslist 是一个可以通过给定字符串选择浏览器版本的工具。

关于 PostCSS、Autoprefixer、postcss-loader、Browserslist 的更多用法，请参考相应的官网文档。

React-Bootstrap

React-Bootstrap 是一个可复用的前端组件库。你可以通过简洁的 React 组件获取 Bootstrap 的外观和体验。

在安装 Bootstrap 后，只需要安装 React-Bootstrap 就可以使用了：

```
npm install react-bootstrap --save
```

这里选取几个常用的组件作为演示。

在Button组件中修改bsStyle属性，就可以创建各种风格的按钮，而在Glyphicon组件中修改glyph属性则可以创建各种图标：

```
<ButtonToolbar>
  <Button>Default</Button>
  <Button bsStyle="primary">Primary</Button>
  <Button bsStyle="success">Success</Button>
  <Button bsStyle="info">Info</Button>
  <Button bsStyle="warning">Warning</Button>
  <Button bsStyle="danger">Danger</Button>
  <Button bsStyle="link">Link</Button>
  <ButtonGroup>
    <Button><Glyphicon glyph="align-left" /></Button>
    <Button><Glyphicon glyph="align-center" /></Button>
    <Button><Glyphicon glyph="align-right" /></Button>
    <Button><Glyphicon glyph="align-justify" /></Button>
  </ButtonGroup>
</ButtonToolbar>
```

还可以快速创建导航栏：

```
<Navbar>
  <Navbar.Header>
    <Navbar.Brand>
      <a href="#">React-Bootstrap</a>
    </Navbar.Brand>
  </Navbar.Header>
  <Nav>
    <NavItem eventKey={1} href="#">Link</NavItem>
    <NavItem eventKey={2} href="#">Link</NavItem>
    <NavDropdown eventKey={3} title="Dropdown" id="basic-nav-dropdown">
      <MenuItem eventKey={3.1}>Action</MenuItem>
      <MenuItem eventKey={3.2}>Another action</MenuItem>
      <MenuItem eventKey={3.3}>Something else here</MenuItem>
      <MenuItem divider />
      <MenuItem eventKey={3.3}>Separated link</MenuItem>
    </NavDropdown>
  </Nav>
</Navbar>
```

难以想象过去满屏幕的 HTML 标签是如何折磨我们的，好在有了 React 后，那些痛苦的日子便一去不复返了。更多的用法，可以参考 React-Bootstrap 官网（http://react-bootstrap.github.io/）。

总结

- bootstrap-loader 是一个用来加载 Bootstrap 的 Webpack 加载器。这个加载器使用 Sass 处理 CSS 样式，对于 Bootstrap 3&4 都能支持。
- PostCSS 是一个使用 JS 插件转换样式的工具。Autoprefixer 是最流行的 PostCSS 插件之一。
- React-Bootstrap 是一个可复用的前端组件库。你可以通过更简洁的 React 组件获取 Bootstrap 的外观和体验。

第四部分

实战篇

第23章

搭建大型项目

本章将介绍如何使用 React 与 Redux 搭建一个可用于生产环境的大型项目。该项目结合了前面章节中所介绍的技术，将现代前端工程中的精华融为一体。这些优秀的技术不仅解放了前端工程师的生产力，而且使得程序本身的性能、功能以及可维护性都得到极大的提升。

例子

本章例子所使用的技术参考了包括 react-redux-universal-hot-example 在内的多个优秀开源项目。所实现的功能如下：

- 第一个页面是主页，用于放置简介信息。
- 第二个页面是计数器页面，用于显示服务器传来的随机数，并可以实现数值的加/减以及重新加载。
- 当找不到路径时将会显示 404 页面，给出页面不存在的提示。
- 程序中一旦发生异步请求，将会在页面中心出现等待图标。

为了实现上述功能，本程序编写了三个服务器：

- 开发服务器，用于在开发环境下提供热替换服务。注意，生产环境下无须启动开发服务器。
- 前端服务器，用于请求数据、渲染页面和代理 API 服务器。

- API 服务器，用于提供 API 服务。

为了在不同环境下完成服务的启动以及资源的构建编译任务，本程序还编写了：

- 两个启动文件，用于在不同环境下启动服务。开发环境将使用 Babel 的 Require Hook 启动服务，生产环境则先编译后启动。
- 两个 Webpack 配置文件，用于在不同环境构建编译客户端资源。

除此之外，本例还使用了 Webpack 同构工具、ESLint 去毛机、bootstrap-loader 等工具，并编写了一些单元测试。

本例的效果如图 23-1、图 23-2 和图 23-3 所示。

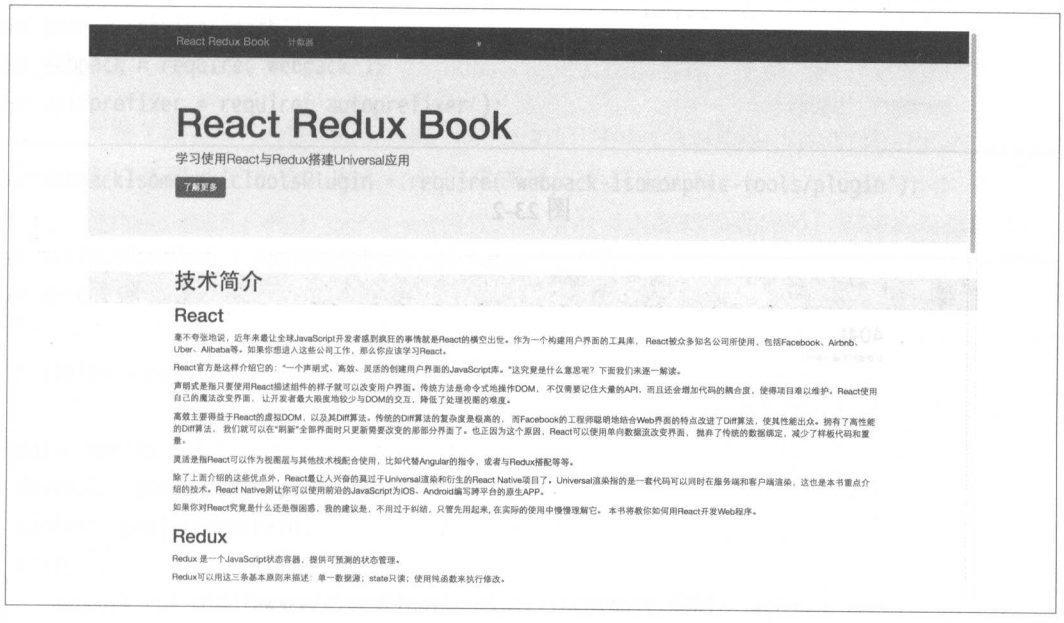


图 23-1

运行方法：

```
npm install
npm start
npm run prod
npm test
```

接下来，让我们一一介绍这些技术。


React Redux Book 计数器
+
-
Increment if odd
Increment async
load
counter的值:
{
"value": 25
}
counter的加载状态:
{
"loading": true,
"loaded": false
}

图 23-2

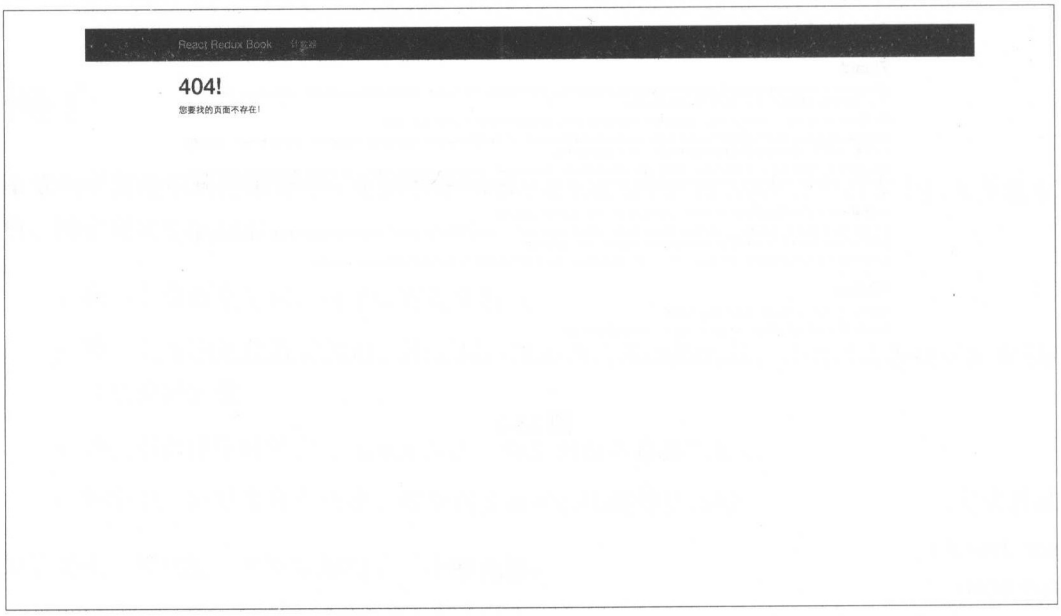

React Redux Book
404!
您要找的页面不存在!

图 23-3

开发服务器

在开发环境中，通常使用开发服务器为程序提供资源服务，实现代码的热替换。开发服务器的使用减少了重复性工作，提高了开发效率。

配置开发服务器，首先需要配置 Webpack。

开发环境下的 Webpack 配置

webpack/dev.config.js

```
var path = require('path');
var webpack = require('webpack');
var autoprefixer = require('autoprefixer');

var WebpackIsomorphicToolsPlugin = require('webpack-isomorphic-tools/plugin');

var projectRootPath = path.resolve(__dirname, '../');
var assetsPath = path.resolve(projectRootPath, './static/dist');

var config = require('../src/config');

module.exports = {
  devtool: 'cheap-eval-source-map',
  context: projectRootPath,
  entry: [
    'webpack-hot-middleware/client?path=http://localhost:3001/__webpack_hmr',
    'bootstrap-loader',
    'font-awesome-loader!./src/theme/font-awesome/font-awesome.config.js',
    './src/client'
  ],
  output: {
    path: assetsPath,
    filename: '[name]-[hash].js',
    chunkFilename: '[name]-[chunkhash].js',
    publicPath: 'http://' + config.host + ':' + (config.port + 1) + '/dist/'
  },
```

```
progress: true,
plugins: [
  new webpack.optimize.OccurrenceOrderPlugin(),
  new webpack.HotModuleReplacementPlugin(),
  new webpack.DefinePlugin({
    __SERVER__: false
  }),
  new WebpackIsomorphicToolsPlugin(require('./webpack-isomorphic-tools')).development
      ()
],
module: {
  loaders: [
    {
      test: /\.js$/,
      loader: 'babel',
      exclude: /node_modules/,
      query: {
        presets: ['react-hmre']
      }
    },
    { test: /\.(jpeg|jpg|png|gif)$/, loader: 'url-loader?limit=10240' },
    {
      test: /\.css$/,
      loaders: [
        'style',
        'css?modules&importLoaders=1&localIdentName=[name]__[local]__[hash:base64:5]',
        'postcss'
      ]
    },
    {
      test: /\.scss$/,
      loaders: [
        'style',
        'css?modules&importLoaders=2&localIdentName=[name]__[local]__[hash:base64:5]',
        'postcss',
        'sass'
      ]
    },
```

```
      {
        test: /\.woff2?(\?v=[0-9]\.[0-9]\.[0-9])?$/,
        loader: "url?limit=10000"
      },
      {
        test: /\.(ttf|eot|svg)(\?[\s\S]+)?$/,
        loader: 'file'
      }
    ]
  },
  postcss: [autoprefixer({ browsers: ['last 2 versions'] })]
};
```

这个文件的大部分配置都与前面章节的相同，但是有些配置做了变动，以适应更高的要求。

1. 开发工具即devtool选项从cheap-module-eval-source-map变为cheap-eval-source-map。两者的重建速度匹敌，但是首次构建速度后者高于前者，所以这里选择了后者。不过cheap-module-eval-source-map在代码映射时可以跳转到源代码，而后者只能跳到编译后的代码。所以如果你更看重代码映射，依然可以使用cheap-module-eval-source-map。这两者都可以很好地用于开发环境。
2. 入口文件增加了'font-awesome-loader!./src/theme/font-awesome/font-awesome.config.js'用于加载 Font Awesome。font-awesome-loader是一个使用 Sass 加载 Font Awesome 的 Webpack 加载器。该工具的用法和 Bootstrap 十分相似，这里不再赘述。
3. 输出文件名增加了哈希值，用于防止缓存。

```
output: {
  path: assetsPath,
  filename: '[name]-[hash].js',
  chunkFilename: '[name]-[chunkhash].js',
  publicPath: 'http://' + config.host + ':' + (config.port + 1) + '/dist/'
}
```

另外，上述代码还使用了配置文件中的host以及port。我们专门编写一个配置文件，用于盛放需要经常变动的变量：

src/config.js

```
module.exports = {
```

```
  host: process.env.HOST || 'localhost',
  port: process.env.PORT || (process.env.NODE_ENV === 'production' ? 8080 : 3000),
  apiHost: process.env.APIHOST || 'localhost',
  apiPort: process.env.APIPORT || 3030,
  app: {
    title: 'React Redux Book',
    description: 'React Redux Book Example',
    head: {
      titleTemplate: 'React Redux Book: %s',
      meta: [
        {
          name: 'description',
          content: 'React Redux Book Example'
        },
        { charset: 'utf-8' }
      ]
    }
  }
};
```

在后面编写前端服务器、API 服务器和组件时，也会用到这个配置文件。因为该文件会输出到客户端，所以切忌将密码等敏感信息填写在这里。

Webpack 同构工具的配置

Webpack 同构工具的配置文件也发生了改变：

webpack/webpack-isomorphic-tools.js

```
var WebpackIsomorphicToolsPlugin = require('webpack-isomorphic-tools/plugin');

var config = {
  assets: {
    images: {extensions: ['png']},
    style_modules: {
      extensions: ['css', 'scss'],
      filter: function (module, regex, options, log) {
        if (options.development) {
```

```
        return WebpackIsomorphicToolsPlugin.style_loader_filter(module, regex, options
            , log);
      } else {
        return regex.test(module.name);
      }
    },
    path: function (module, options, log) {
      if (options.development) {
        return WebpackIsomorphicToolsPlugin.style_loader_path_extractor(module,
            options, log);
      } else {
        return module.name;
      }
    },
    parser: function (module, options, log) {
      if (options.development) {
        return WebpackIsomorphicToolsPlugin.css_modules_loader_parser(module, options,
            log);
      } else {
        return module.source;
      }
    }
  }
 }
};

module.exports = config;
```

与之前相比，这里增加了对options.development的条件判断，如果条件成立，则代表处于开发环境，将返回适用于style-loader和css-loader的返回值；如果条件不成立，则代表处于生产环境，因为生产环境不再使用style-loader，而是提取为单独的样式文件，所以只需要返回默认值即可。

独立的开发服务器

为了不让前端服务器阻碍开发服务器的运行，这里将开发服务器独立出来：

webpack/webpack-dev-server.js

```
var Express = require('express');

var webpack = require('webpack');
var webpackDevMiddleware = require('webpack-dev-middleware');
var webpackHotMiddleware = require('webpack-hot-middleware');
var webpackConfig = require('./dev.config');

var app = new Express();
var port = require('../src/config').port + 1;

var compiler = webpack(webpackConfig);
app.use(webpackDevMiddleware(compiler, {noInfo: true, publicPath: webpackConfig.output.
    publicPath}));
app.use(webpackHotMiddleware(compiler));

app.listen(port, (error) => {
  if (error) {
    console.error(error)
  } else {
    console.info(`==> 🚧  Webpack development server listening on port ${port}.`)
  }
});
```

开发服务器的写法和之前一样。不过，注意端口号需要和 Webpack 中的保持一致，这里统一为port + 1。

启动开发服务器

通常在 package.json 中编写 NPM 脚本，来启动开发服务器：

```
"watch-client": "node webpack/webpack-dev-server"
```

将脚本命名为watch-client，表示监视客户端变动，在port + 1端口开启服务，随时给客户端提供资源，并在代码变动时进行热替换。

前端服务器

前端服务器用于渲染 React 组件并吐出多个页面，它使用了多种技术，包括 React 组件的服务端渲染、React Router 的服务端路由、Webpack 同构渲染、数据预载和延迟渲染、代理 API 服务器、压缩等。

配置前端服务器

src/server.js

```
import path from 'path';
import Express from 'express';
import favicon from 'serve-favicon';
import httpProxy from 'http-proxy';
import compression from 'compression';
import React from 'react';
import { renderToString } from 'react-dom/server';
import { Provider } from 'react-redux';
import { match, RouterContext } from 'react-router';
import configureStore from './utils/configureStore';
import getRoutes from './routes';
import Html from './utils/Html';
import config from './config';

const app = new Express();
const port = config.port;
const targetUrl = 'http://' + config.apiHost + ':' + config.apiPort;
const proxy = httpProxy.createProxyServer({
  target: targetUrl
});

app.use(compression());
app.use(Express.static(path.join(__dirname, '..', 'static')));
app.use(favicon(path.join(__dirname, '..', 'static', 'favicon.ico')));

app.use('/api', (req, res) => {
```

```
  proxy.web(req, res, { target: targetUrl });
});

app.use((req, res) => {
  if (process.env.NODE_ENV !== 'production') {
    webpackIsomorphicTools.refresh();
  }

  const store = configureStore();
  const routes = getRoutes(store);

  function hydrateOnClient() {
    res.send('<!doctype html>\n' +
      renderToString(<Html assets={webpackIsomorphicTools.assets()} store={store}/>));
  }

  if (__DISABLE_SSR__) {
    hydrateOnClient();
    return;
  }

  match({ routes, location: req.url }, (err, redirect, renderProps) => {
    if (redirect) {
      res.redirect(redirect.pathname + redirect.search);
    } else if (err) {
      res.status(500);
      hydrateOnClient();
      console.error('ROUTER ERROR:', err.stack);
    } else if (renderProps) {
      res.status(200);
      const component = (
        <Provider store={store}>
          <RouterContext {...renderProps} />
        </Provider>
      );
      res.send('<!doctype html>\n' +
        renderToString(
```

```
        <Html assets={webpackIsomorphicTools.assets()} component={component} store={
            store}/>)
      );
    } else {
      res.status(404).send('Not found');
    }
  });
});

app.listen(port, (error) => {
  if (error) {
    console.error(error);
  } else {
    console.info('==> 🌎  Open http://%s:%s in a browser to view the app.', config.host,
        port);
  }
});
```

前端服务器的大部分配置与前面章节介绍的相同，但是这里针对生产环境做了一些优化。

1. 将端口号写在配置文件中，方便了部署时端口变动。

   ```
   const port = config.port;
   ```

2. 使用http-proxy，将 API 服务器代理到前端服务器，主要用于解决跨域的问题，这里也可以直接使用代理服务器。

   ```
   const targetUrl = 'http://' + config.apiHost + ':' + config.apiPort;
   const proxy = httpProxy.createProxyServer({
     target: targetUrl
   });

   ...

   app.use('/api', (req, res) => {
     proxy.web(req, res, { target: targetUrl });
   });
   ```

3. 采用 Express 的压缩拓展启动压缩，支持 deflate 和 gzip 两种压缩方式。

   ```
   app.use(compression());
   ```

4. 编写 React 组件 Html 替代原来的模板字符串。
5. 编写hydrateOnClient函数用于禁止服务端渲染。

```
function hydrateOnClient() {
  res.send('<!doctype html>\n' +
    renderToString(<Html assets={webpackIsomorphicTools.assets()} store={store
        }/>));
}

if (__DISABLE_SSR__) {
  hydrateOnClient();
  return;
}
```

使用组件渲染 HTML 页面

因为多页面的管理比较复杂，简单的模板字符串已经不能满足需求。所以，这里编写一个 Html 组件用于渲染 HTML 页面。因为它是无状态函数，而且不同于普通组件，这里将其放在工具集目录（src/utils）下。



注意，Utils和Helpers都可以表示工具集和辅助模块，但是Helpers中的模块通常需要实例化，而Utils则放置一些函数。因此，不仅是 Html 组件，这里将配置 store 的函数以及常用的工具函数都放在 src/utils 下面。不过，目录的命名要求并不严格，可以根据自己和团队习惯灵活安排目录结构。

Html 组件代码如下：

src/utils/Html.js

```
import React, { PropTypes } from 'react';
import ReactDOM from 'react-dom/server';
import serialize from 'serialize-javascript';
import Helmet from 'react-helmet';

function Html(props) {
  const { assets, component, store } = props;
  const content = component ? ReactDOM.renderToString(component) : '';
```

```
  const head = Helmet.rewind();

  return (
    <html lang="zh-CN">
      <head>
        {head.base.toComponent()}
        {head.title.toComponent()}
        {head.meta.toComponent()}
        {head.link.toComponent()}
        {head.script.toComponent()}

        <link rel="shortcut icon" href="/favicon.ico" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        {Object.keys(assets.styles).map((style, key) =>
          <link
            href={assets.styles[style]} key={key} media="screen, projection"
            rel="stylesheet" type="text/css"
            charSet="UTF-8"
          />
        )}
      </head>
      <body>
        <div id="app" dangerouslySetInnerHTML={{ __html: content }} />
        <script
          dangerouslySetInnerHTML={{
            __html: `window.__INITIAL_STATE__=${serialize(store.getState())};`
          }}
          charSet="UTF-8"
        />
        <script src={assets.javascript.main} charSet="UTF-8" />
      </body>
    </html>
    );
}

Html.propTypes = {
  assets: PropTypes.object,
  component: PropTypes.node,
```

```
  store: PropTypes.object
};

export default Html;
```

这个组件做了以下几件事：

1. 接收 Webpack 同构工具提供的键值对对象、React 组件component以及 Redux 的 store 作为 props。

   ```
   const { assets, component, store } = props;
   ```

2. 使用react-helmet管理文档头信息，包括 title、meta、link、script 和 base tags。

   ```
   const head = Helmet.rewind();

   ...

   <head>
      {head.base.toComponent()}
      {head.title.toComponent()}
      {head.meta.toComponent()}
      {head.link.toComponent()}
      {head.script.toComponent()}

            ...
   </head>
   ```

3. 通过遍历 Webpack 同构工具提供的资源键值对对象，给 HTML 添加独立出来的样式文件。注意，只有在生产环境下，assets.styles才包含数据。

   ```
   {Object.keys(assets.styles).map((style, key) =>
     <link
       href={assets.styles[style]} key={key} media="screen, projection"
       rel="stylesheet" type="text/css"
       charSet="UTF-8"
     />
   )}
   ```

4. 使用dangerouslySetInnerHTML渲染 HTML 字符串到组件中。默认情况下，React 会对 HTML 进行编码来防止 XSS 攻击。如果你实在想渲染 HTML 字符串，应该使用 dan-

gerouslySetInnerHTML。React 故意使用这个笨重、醒目的语法来让开发者不会意外渲染文本作为 HTML，继而导致 XSS 漏洞。

```
<div id="app" dangerouslySetInnerHTML={{ __html: content }} />
```

5. 渲染来自服务端的预载 state 和用于客户端渲染的脚本文件。serialize用于将 JSON 转换为字符串，它和JSON.stringify()的区别在于可以输出 JSON 中的函数和正则表达式。另外，它也包含自动编码功能来防止 XSS 攻击。

```
<script
  dangerouslySetInnerHTML={{
    __html: `window.__INITIAL_STATE__=${serialize(store.getState())};`
  }}
  charSet="UTF-8"
/>
<script src={assets.javascript.main} charSet="UTF-8" />
```

启动前端服务器

为了在不同环境下启动前端服务器，这里专门编写了启动文件：

bin/server.js

```
/* eslint-disable */
var path = require('path');
var rootDir = path.resolve(__dirname, '..');

var WebpackIsomorphicTools = require('webpack-isomorphic-tools');

global.__SERVER__ = true;
global.__DISABLE_SSR__ = false;

if (process.env.NODE_ENV === 'production') {
  global.webpackIsomorphicTools = new WebpackIsomorphicTools(
    require('../webpack/webpack-isomorphic-tools'))
    .server(rootDir, function () {
      require('../build/server');
    });
}
else {
```

```
  require('babel-register');
  global.webpackIsomorphicTools = new WebpackIsomorphicTools(
    require('../webpack/webpack-isomorphic-tools'))
    .development()
    .server(rootDir, function () {
      require('../src/server');
    });
}
```

这个启动文件配置了 Webpack 同构工具，定义了一些全局变量，并根据环境的不同进行相应的配置。

最后使用 NPM 命令启动前端服务器。在开发环境下，使用nodemon替代node，在每次src文件夹中的文件发生变动时，能自动重启，提高了开发效率。

```
"start-prod": "cross-env NODE_ENV=production node bin/server",
"start-dev": "nodemon  --watch src bin/server.js",
```

至此前端服务器就搭建完成了。

API 服务器

Universal 渲染的服务端指的是前端服务器，尽管可以在前端服务器中进行读/写数据库、用户认证等，但是由于前端服务器和客户端公用一套代码，所以不管出于安全的考虑，还是因为模块分离方面的因素，都应该将 API 服务器独立出去。API 服务器不限定必须使用 Node.js 编写，不过这里不仅选择了 Node.js，还使用了 ES2015 的语法和新特性。

配置 API 服务器

API 服务的编写并不是本书的重点，我们仅用它做测试，所以这里只编写了一些控制器，并监听由配置文件所指定的端口。如果在真实项目中，还需要做更多的配置。

src/api/api.js

```
import Express from 'express';
import config from '../config';
import controllers from './controllers';
```

```
const port = config.apiPort;
const app = new Express();

controllers(app);

app.listen(port, (error) => {
  if (error) {
    console.error(error);
  } else {
    console.info('==> 🌎  API Listening on port %s. ', port);
  }
});
```

启动 API 服务器

为了在不同环境下启动 API 服务器，这里专门编写了启动文件：

bin/api.js

```
/* eslint-disable */
if (process.env.NODE_ENV === 'production') {
  require('../build/api/api');
}
else {
  require('babel-register');
  require('../src/api/api');
}
```

这个启动文件无须配置 Webpack 同构工具，也没有定义全局变量，仅仅只是根据不同环境进行相应的配置。

最后使用 NPM 命令启动 API 服务器。

```
"start-api-dev": "nodemon  --watch src/api bin/api.js",
"start-api-prod": "cross-env NODE_ENV=production node bin/api",
```

生产环境下的构建编译

在前面的章节中，所有的示例程序都只做了开发环境下的构建编译。例如，使用 Require Hook 来运行需要 Babel 编译的 Node.js 程序，使用开发服务器打包客户端的各种资源，如脚本、样式、图片等。但是这些构建编译方法只是方便了开发，在生产环境下会引起包括内存消耗过大在内的各方面问题。那么如何在生产环境下做构建编译工作呢？接下来我们一一讲解。

编译运行 Node.js

在生产环境下，应该先使用 Babel 编译 Node.js 程序，然后使用node运行。

首先，删除上次编译的老旧文件。rimraf是 Node.js 的一个删除工具，相当于rm -rf。这里不仅删除了build文件夹，也删除了 Webpack 的目标路径，即static/dist文件夹。

```
"clean": "rimraf build static/dist"
```

然后，使用 Babel 编译。src代表源文件目录，-d build代表目标路径为build，--copy-files代表将非 JavaScript 文件也拷贝到目标文件夹中。

```
"build-server": "babel src -d build --copy-files"
```

最后，直接用node运行。

```
"start-prod": "cross-env NODE_ENV=production node bin/server",
...
"start-api-prod": "cross-env NODE_ENV=production node bin/api"
```

注意，在生产环境下，入口文件已经发生了改变：

bin/server.js

```
...
if (process.env.NODE_ENV === 'production') {
  global.webpackIsomorphicTools = new WebpackIsomorphicTools(
    require('../webpack/webpack-isomorphic-tools'))
    .server(rootDir, function () {
      require('../build/server');
    });
}
...
```

bin/api.js

```
...
if (process.env.NODE_ENV === 'production') {
  require('../build/api/api');
}
...
```

而开发环境依然使用 Require Hook，这里不再赘述。

生产环境下的 Webpack 配置

在生产环境中，不需要使用开发服务器来提供资源，而是直接将其打包到静态资源目录，然后在页面中引入入口文件即可。当然，Webpack 的配置也与开发环境不同：

webpack/prod.config.js

```
var path = require('path');
var webpack = require('webpack');
var WebpackIsomorphicToolsPlugin = require('webpack-isomorphic-tools/plugin');
var ExtractTextPlugin = require('extract-text-webpack-plugin');
var autoprefixer = require('autoprefixer');

var projectRootPath = path.resolve(__dirname, '../');
var assetsPath = path.resolve(projectRootPath, './static/dist');

module.exports = {
  devtool: 'source-map',
  context: projectRootPath,
  entry: [
    'bootstrap-loader/extractStyles',
    'font-awesome-loader!./src/theme/font-awesome/font-awesome.config.prod.js',
    './build/client'
  ],
  output: {
    path: assetsPath,
    filename: '[name]-[chunkhash].js',
    chunkFilename: '[name]-[chunkhash].js',
```

```
    publicPath: '/dist/'
  },
  progress: true,
  plugins: [
    new ExtractTextPlugin('[name]-[chunkhash].css', { allChunks: true }),
    new webpack.optimize.DedupePlugin(),
    new webpack.optimize.OccurrenceOrderPlugin(),
    new webpack.optimize.UglifyJsPlugin({
      compress: {
        warnings: false
      }
    }),
    new webpack.DefinePlugin({
      'process.env': {
        NODE_ENV: '"production"'
      },
      __SERVER__: false
    }),
    new WebpackIsomorphicToolsPlugin(require('./webpack-isomorphic-tools'))
  ],
  module: {
    loaders: [
      { test: /\.(jpeg|jpg|png|gif)$/, loader: 'url-loader?limit=10240' },
      {
        test: /\.css$/,
        loader: ExtractTextPlugin.extract(
          'style',
          'css?modules&importLoaders=1&localIdentName=[name]__[local]__[hash:base64:5]'
              +
          '!postcss'
        )
      },
      {
        test: /\.scss$/,
        loader: ExtractTextPlugin.extract(
          'style',
          'css?modules&importLoaders=2&localIdentName=[name]__[local]__[hash:base64:5]'
              +
```

```
        '!postcss' +
        '!sass'
      )
    },
    {
      test: /\.woff2?(\?v=[0-9]\.[0-9]\.[0-9])?$/,
      loader: "url?limit=10000"
    },
    {
      test: /\.(ttf|eot|svg)(\?[\s\S]+)?$/,
      loader: 'file'
    }
  ]
 },
 postcss: [autoprefixer({ browsers: ['last 2 versions'] })]
};
```

与开发环境相比，发生了以下变动：

1. devtool设置为source-map，这种模式构建速度非常慢，但是会生成完整的源代码映射，适用于生产环境。
2. 在入口文件列表中去除了热替换文件，在 bootstrap 后面添加了extractStyles，用于提取出独立的文件。Font Awesome 的配置文件和客户端入口都发生了变化。

```
entry: [
    'bootstrap-loader/extractStyles',
    'font-awesome-loader!./src/theme/font-awesome/font-awesome.config.prod.js',
    './build/client'
  ]
```

3. output.publicPath从开发服务器的服务路径改为了/dist/，代表资源文件从前端服务器中加载。
4. 插件列表增加了 ExtractTextPlugin，用于提取样式文件，还添加了 DedupePlugin 和 UglifyJsPlugin 用于去重和压缩。另外，去除了 HotModuleReplacementPlugin 插件，取消了 Webpack 同构工具的开发模式。

```
plugins: [
  new ExtractTextPlugin({ filename: '[name]-[chunkhash].css', allChunks: true }),
  new webpack.optimize.DedupePlugin(),
```

```
    new webpack.optimize.OccurrenceOrderPlugin(),
    new webpack.optimize.UglifyJsPlugin({
      compress: {
        warnings: false
      }
    }),
    new webpack.DefinePlugin({
      'process.env': {
        NODE_ENV: '"production"'
      },
      __SERVER__: false
    }),
    new WebpackIsomorphicToolsPlugin(require('./webpack-isomorphic-tools'))
  ]
```

5. 加载器列表移除了对 JavaScript 文件的处理，因为使用build/client作为入口的话，就无须再使用 Babel 进行编译了。样式的加载器均添加了ExtractTextPlugin的配置。

```
  loaders: [
    { test: /\.(jpeg|jpg|png|gif)$/, loader: 'url-loader?limit=10240' },
    {
      test: /\.css$/,
      loader: ExtractTextPlugin.extract(
        'style',
        'css?modules&importLoaders=1&localIdentName=[name]__[local]__[hash:base64
            :5]' +
        '!postcss'
      )
    },
    {
      test: /\.scss$/,
      loader: ExtractTextPlugin.extract(
        'style',
        'css?modules&importLoaders=2&localIdentName=[name]__[local]__[hash:base64
            :5]' +
        '!postcss' +
        '!sass'
      )
    },
```

```
      {
        test: /\.woff2?(\?v=[0-9]\.[0-9]\.[0-9])?$/,
        loader: "url?limit=10000"
      },
      {
        test: /\.(ttf|eot|svg)(\?[\s\S]+)?$/,
        loader: 'file'
      }
    ]
```

接下来，编写 NPM 命令启动客户端资源的编译：

```
"build-client": "webpack --config webpack/prod.config.js",
```

至此，生产环境下的构建编译就配置完成了。

最后，需要编写两个 NPM 命令，用于在不同环境下启动所有的构建编译任务。

```
"prod": "npm run build && concurrently -k \"npm run start-api-prod\" \"npm run start-
    prod\"",
"dev": "concurrently -k \"npm run watch-client\" \"npm run start-api-dev\" \"npm run
    start-dev\"",
```

其中，concurrently可以让多条命令平行启动。更多的用法，请参考官网文档（https://github.com/kimmobrunfeldt/concurrently）。

公用代码

在 src 目录中，除了 src/api、src/server.js 和 src/client.js 外，其他都是“公用代码”，包括工具集目录、路由配置、容器页面和组件等。

工具集

为了目录的简约，这里将 Html 组件函数、配置 store 的函数以及其他工具函数都放在了 src/utils 目录下面，因为它们都只是“工具函数”，并不涉及业务逻辑。Html 组件函数在前面已经介绍过了，下面一一讲解其他几个函数。

配置 store 的函数与前面章节有所不同，这里主要是在生产环境中移除了开发者工具：

src/utils/configureStore.js

```
import { createStore, applyMiddleware, compose } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from '../reducers';
import { asyncMiddleware } from 'redux-amrc';

let createStoreWithMiddleware;

if (process.env.NODE_ENV === 'production') {
  createStoreWithMiddleware = compose(
    applyMiddleware(thunk, asyncMiddleware),
  )(createStore);
} else {
  createStoreWithMiddleware = compose(
    applyMiddleware(thunk, asyncMiddleware),
    typeof window === 'object' &&
    typeof window.devToolsExtension !== 'undefined' ? window.devToolsExtension() : f =>
        f
  )(createStore);
}

export default function configureStore(initialState) {
  const store = createStoreWithMiddleware(rootReducer, initialState);

  if (module.hot) {
    // Enable Webpack hot module replacement for reducers
    module.hot.accept('../reducers', () => {
      const nextRootReducer = require('../reducers/index').default;
      store.replaceReducer(nextRootReducer);
    });
  }

  return store;
}
```

其他工具函数放在 src/utils/utils.js 中。本例只有一个customFetch方法，它能够根据运行环境的不同来选择 URL 前缀，而且还封装了错误处理的逻辑。我们将使用customFetch替代fetch方法进行网络请求。

src/utils/utils.js

```
import 'isomorphic-fetch';
import config from '../config';

function handleErrors(res) {
  if (!res.ok) {
    throw new Error(res.statusText);
  }
  return res.json();
}

export function customFetch(url, option) {
  const prefix = __SERVER__ ? 'http://' + config.apiHost + ':' + config.apiPort : '/api
      ';

  const opt = option || {};

  return fetch(prefix + url, opt)
    .then(handleErrors);
}
```

在你自己的项目中，可以将更多的工具函数放在这个目录下。

路由与页面

路由与页面涉及业务逻辑，本章的示例程序包含三个页面，包括主页、计数器和 404 页面，它们都在路由中进行配置。

src/routes.js

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';
import { Main, Home, Counter, NotFound, Forms, Statistic, Login } from './containers';
import { loadCounter } from './actions/counter';
```

```
import { loadStatistic } from './actions/statistic';
import { loadAuthIfNeeded } from './actions/auth';

const preload = promise => (nextState, replace, cb) => {
  if (__SERVER__ || nextState.location.action === 'PUSH') {
    promise().then(() => cb());
  } else {
    cb();
  }
};

export default store => {
  const counterPromise = () => store.dispatch(loadCounter());
  return (
    <Route path="/" component={Main}>
      <IndexRoute component={Home}/>
      <Route path="counter" component={Counter} onEnter={preload(counterPromise)}/>
      <Route path="*" component={NotFound} status={404}/>
    </Route>
  );
};
```

Home、Counter、NotFound 分别就是主页、计数器和 404 页面。Main 组件则是所有页面公用的父级页面。

src/containers/Main/Main.js

```
import React, { Component, PropTypes } from 'react';
import { IndexLink } from 'react-router';
import { Navbar, Nav, NavItem } from 'react-bootstrap';
import { LinkContainer } from 'react-router-bootstrap';
import Helmet from 'react-helmet';
import { Spin } from '../../components';
import config from '../../config';

class Main extends Component { //eslint-disable-line
  static propTypes = {
    children: PropTypes.any.isRequired
  };
```

```
  render() {
    require('./Main.scss');
    return (
      <div>
        <Helmet {...config.app.head}/>
        <Navbar inverse>
          <Navbar.Header>
            <Navbar.Brand>
              <IndexLink to="/">
                <span>{config.app.title}</span>
              </IndexLink>
            </Navbar.Brand>
            <Navbar.Toggle />
          </Navbar.Header>

          <Navbar.Collapse>
            <Nav navbar>
              <LinkContainer to="/counter">
                <NavItem eventKey={1}>计数器</NavItem>
              </LinkContainer>
            </Nav>
          </Navbar.Collapse>
        </Navbar>

        <Spin />

        <div>
          {/* this will render the child routes */}
          {React.cloneElement(this.props.children, this.props)}
        </div>
      </div>
    );
  }
}

export default Main;
```

和前面章节相比，这里的 Main 组件增加了以下几个功能：

1. 使用了 Main.scss 配置样式，其中:global()语法用于取消默认的 CSS Module。

```
:global(.navbar) {
  margin-bottom: 0;
  border-radius: 0;
}
```

2. 使用<Helmet {...config.app.head}/>为程序添加文档头信息。信息内容来自配置文件。
3. 使用了 React-Bootstrap 提供的导航栏组件，并配置了页面的标题。
4. 通过监听state.async.loadingNumber，在发生异步事件时显示等待图标。为了不让异步请求先后的状态变化影响全局，这里将等待图标封装成一个单独容器组件Spin。如此一来，异步请求先后的状态变化便只会重新渲染等待图标组件Spin了。

src/components/Spin/Spin.js

```
import React, { PropTypes } from 'react';
import { connect } from 'react-redux';

function Spin(props) {
  const styles = require('./Spin.scss');
  return (
    <div className={styles.spin}>
      {props.loadingNumber > 0 && (
        <i className="fa fa-spinner fa-spin fa-3x fa-fw"/>
      )}
    </div>
  );
}

Spin.propTypes = {
  loadingNumber: PropTypes.number.isRequired
};

export default connect(
  state => ({ loadingNumber: state.async.loadingNumber || 0 })
)(Spin);
```

src/components/Spin/Spin.scss

```
.spin{
  position:fixed;
  top:50%;
  left:50%;
  transform: translate(-50%, -50%);
  color: #9d9d9d;
  z-index: 999;
}
```

主页只是渲染一些简介信息，计数器页面与之前并无二致，这里不再具体讲解。不过注意，两个页面都使用 Helmet 组件配置标题。

质量保证

质量保证的重要性和必要性不言而喻。常见的质量保证技术包括：测试、去毛、持续集成等。关于持续集成的相关知识，会在后面章节详细讲解。本章主要探讨测试和去毛这两种技术。

测试

虽然测试的相关知识已经在前面的章节讲解过了，但是由于 Webpack 同构工具的使用以及网络请求的使用，这里还要再补充一些测试的技巧。

Webpack 同构工具是通过生成 webpack-assets.json 文件，来给`require()`方法打补丁的。因此，在运行测试前，必须要先生成这个文件。这里使用了`pretest`命令来生成 webpack-assets.json 文件，这个命令会在`npm test`命令运行前运行。

```
"pretest": "webpack --config webpack/dev.config.js"
```

生成 webpack-assets.json 文件后，接下来还需要在运行测试前，配置 Webpack 同构工具。因为我们的测试脚本运行在 Node.js 中，因此配置方法和前面介绍的服务端调用方法类似。另外，为了在运行测试前配置 Webpack 同构工具，这里还使用了 Mocha 的一个全局函数`before()`。这个函数会在运行测试之前运行，当它的参数`done`被调用后，才会执行后续的测试。最后，为了防止在 watch 模式下，重复实例化`WebpackIsomorphicTools`，我们还在实例化`WebpackIsomorphicTools`之前进行了条件判断。

test/components/Counter.spec.js

```
before(done => {
  if (typeof webpackIsomorphicTools !== 'undefined') {
    done();
    return;
  }
  const WebpackIsomorphicTools = require('webpack-isomorphic-tools');
  global.webpackIsomorphicTools = new WebpackIsomorphicTools(
    require('../../webpack/webpack-isomorphic-tools'))
    .development()
    .server(path.resolve(__dirname, '..', '..'), done);
});
```

经过上述配置，就可以通过require()方法调用组件了，而且即便你在组件中使用了 Webpack 同构工具，也不会出错。

另外，为了在测试包含网络请求的容器组件，本例还使用了 nock 这个工具来模拟 API。

test/containers/Counter.spec.js

```
import expect from 'expect';
import React from 'react';
import { mount } from 'enzyme';
import { Provider } from 'react-redux';
import nock from 'nock';
import { Counter } from '../../src/containers';
import { loadCounter } from '../../src/actions/counter';
import configureStore from '../../src/utils/configureStore';
import config from '../../src/config';

function setup(value = 0) {
  global.__SERVER__ = true;
  nock('http://' + config.apiHost + ':' + config.apiPort)
    .get('/counter')
    .reply(200, { value });

  const store = configureStore();
  return store.dispatch(loadCounter())
```

```
    .then(() => {
      const app = mount(
        <Provider store={store}>
          <Counter />
        </Provider>
      );
      return {
        buttons: app.find('button'),
        pre: app.find('pre').at(0)
      };
    });
}

describe('containers', () => {
  describe('Counter', () => {
    afterEach(() => {
      nock.cleanAll();
    });

    it('should display initial count', () => setup()
      .then(({ pre }) => {
        expect(pre.text()).toMatch(/"value": 0/);
      }));

    it('should display updated count after increment button click', () => setup()
      .then(({ buttons, pre }) => {
        buttons.at(0).simulate('click');
        expect(pre.text()).toMatch(/"value": 1/);
      }));

    it('should display updated count after decrement button click', () => setup()
      .then(({ buttons, pre }) => {
        buttons.at(1).simulate('click');
        expect(pre.text()).toMatch(/"value": -1/);
      }));

    it('shouldnt change if even and if odd button clicked', () => setup()
      .then(({ buttons, pre }) => {
```

```
      buttons.at(2).simulate('click');
      expect(pre.text()).toMatch(/"value": 0/);
    }));

  it('should change if odd and if odd button clicked', () => setup(1)
    .then(({ buttons, pre }) => {
      buttons.at(2).simulate('click');
      expect(pre.text()).toMatch(/"value": 2/);
    }));

  it('should change if incrementAsync button clicked', done => {
    setup(1).then(({ buttons, pre }) => {
      buttons.at(3).simulate('click');
      setTimeout(() => {
        expect(pre.text()).toMatch(/"value": 2/);
        done();
      }, 1000);
    });
  });

  it('should display updated count after load button click', done => {
    setup(1)
      .then(({ buttons, pre }) => {
        nock.cleanAll();
        nock('http://' + config.apiHost + ':' + config.apiPort)
          .get('/counter')
          .reply(200, { value: 100 });
        buttons.at(4).simulate('click');
        setTimeout(() => {
          expect(pre.text()).toMatch(/"value": 100/);
          done();
        }, 1000);
      });
  });
 });
});
```

关于 nock 的更多用法，请参考它的官网文档（https://github.com/node-nock/nock）。

Airbnb 编码规范

除了测试之外，去毛也是质量保证的常用技术。去毛技术往往与编码规范紧密关联，下面我们就先来了解一下著名的 Airbnb 编码规范。

Airbnb 的编码规范素来广受好评，以至于 Redux 官方本没有用 Airbnb 编码规范，但是为了符合大多数人的喜好和习惯，专门开启了一个符合 Airbnb 编码规范的分支。本书所有的示例程序都采用了 Airbnb 规范。它不仅规定了一些良好的代码书写习惯，还集成了一些 React 开发的最佳实践，这里列举一部分 React 规则。

基本规则

- 每个文件只包含一个 React 组件。但是无状态组件允许一个文件包含多个组件。
- 始终使用 JSX 语法。
- 除非从一个非 JSX 文件中初始化 app，否则不要使用`React.createElement`。

Class vs React.createClass vs stateless

- 除非有更好的理由使用混淆（mixins），否则就使用组件类继承`React.Component`。

```
// bad
const Listing = React.createClass({
  render() {
    return <div />;
  }
});

// good
class Listing extends React.Component {
  render() {
    return <div />;
  }
}
```

- 如果没有组件没有内部状态，那么普通函数（不要使用箭头函数）比类的写法更好：

```
// bad
class Listing extends React.Component {
  render() {
```

```
    return <div>{this.props.hello}</div>;
  }
}

// bad (因为箭头函数没有“name”属性)
const Listing = ({ hello }) => (
  <div>{hello}</div>
);

// good
function Listing({ hello }) {
  return <div>{hello}</div>;
}
```

命名

- **扩展名**：使用.jsx作为 React 组件的扩展名。
- **文件名**：文件命名采用帕斯卡命名法，如ReservationCard.jsx。
- **引用名**：组件引用采用帕斯卡命名法，其实例采用驼峰式命名法。

 注意，因为社区公认使用.jsx作为扩展名已经是过时的做法，所以本书没有使用这个规则。

```
// bad
const reservationCard = require('./ReservationCard');

// good
const ReservationCard = require('./ReservationCard');

// bad
const ReservationItem = <ReservationCard />;

// good
const reservationItem = <ReservationCard />;
```

- **组件命名**：使用文件名作为组件名。例如，ReservationCard.jsx组件的引用名应该是ReservationCard。然而，对于一个目录的根组件，应该使用index.jsx作为文件名，使用目录名作为组件名。

```
// bad
const Footer = require('./Footer/Footer.jsx')

// bad
const Footer = require('./Footer/index.jsx')

// good
const Footer = require('./Footer')
```

声明

不要通过displayName来命名组件，而要通过引用来命名组件。

```
// bad
export default React.createClass({
  displayName: 'ReservationCard',
  // stuff goes here
});

// good
export default class ReservationCard extends React.Component {
}
```

对齐

对于 JSX 语法，遵循下面的对齐风格：

```
// bad
<Foo superLongParam="bar"
     anotherSuperLongParam="baz" />

// good
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
/>

// 如果组件的属性可以放在一行就保持在当前一行中
<Foo bar="bar" />
```

```
// 多行属性采用缩进
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
>
  <Spazz />
</Foo>
```

引号

对于 JSX 使用双引号，对其他所有 JavaScript 属性使用单引号。

```
// bad
<Foo bar='bar' />

// good
<Foo bar="bar" />

// bad
<Foo style={{ left: "20px" }} />

// good
<Foo style={{ left: '20px' }} />
```

空格

在自闭合标签之前留一个空格。

```
// bad
<Foo/>

// very bad
<Foo                 />

// bad
<Foo
 />

// good
<Foo />
```

属性

属性名采用驼峰式命名法。

```
// bad
<Foo
  UserName="hello"
  phone_number={12345678}
/>

// good
<Foo
  userName="hello"
  phoneNumber={12345678}
/>
```

括号

当组件跨行时，要用括号包裹 JSX 标签。

```
// bad
render() {
  return <MyComponent className="long body" foo="bar">
           <MyChild />
         </MyComponent>;
}

// good
render() {
  return (
    <MyComponent className="long body" foo="bar">
      <MyChild />
    </MyComponent>
  );
}

// good, when single line
render() {
  const body = <div>hello</div>;
  return <MyComponent>{body}</MyComponent>;
}
```

标签

- 没有子组件的父组件使用自闭合标签。

```
// bad
<Foo className="stuff"></Foo>

// good
<Foo className="stuff" />
```

- 如果组件有多行属性，闭合标签应写在新的一行上。

```
// bad
<Foo
  bar="bar"
  baz="baz" />

// good
<Foo
  bar="bar"
  baz="baz"
/>
```

方法

React 组件的内部方法命名不要使用下画线前缀。

```
// bad
React.createClass({
  _onClickSubmit() {
    // do stuff
  }

  // other stuff
});

// good
class extends React.Component {
  onClickSubmit() {
    // do stuff
  }
```

```
  // other stuff
});
```

顺序

- 继承 React.Component 的类的方法时遵循下面的顺序：
 1. constructor
 2. optional static methods
 3. getChildContext
 4. componentWillMount
 5. componentDidMount
 6. componentWillReceiveProps
 7. shouldComponentUpdate
 8. componentWillUpdate
 9. componentDidUpdate
 10. componentWillUnmount
 11. 点击回调或者事件回调，比如 onClickSubmit() 或者 onChangeDescription()
 12. render 函数中的 getter 方法，比如 getSelectReason() 或者 getFooterContent()
 13. 可选的 render 方法，比如 renderNavigation() 或者 renderProfilePicture()
 14. render
- 定义 propTypes、defaultProps、contextTypes 等。

```
import React, { PropTypes } from 'react';

const propTypes = {
  id: PropTypes.number.isRequired,
  url: PropTypes.string.isRequired,
  text: PropTypes.string,
};

const defaultProps = {
  text: 'Hello World',
};
```

```
class Link extends React.Component {
  static methodsAreOk() {
    return true;
  }

  render() {
    return <a href={this.props.url} data-id={this.props.id}>{this.props.text}</a>
  }
}

Link.propTypes = propTypes;
Link.defaultProps = defaultProps;

export default Link;
```

以上只是关于 React 的编码规范，另外还有关于 JavaScript、样式的规范等，这里不再赘述。注意，以上规则并非“死规定”，可以根据自身以及团队习惯进行修改和拓展。

去毛机

开启 Airbnb 规范的最好办法就是在去毛机中集成。去毛机能够检查源代码中的错误以及不规范之处，提醒开发者及时修改，以此来保证代码质量。优秀的去毛机有很多，这里使用了 ESLint。

- 安装

```
npm install -g eslint
```

- 使用

```
eslint --init
```

还可以通过配置文件配置规则，忽略一些目录：

.eslintrc.json

```
{
  "extends": "airbnb",
  "env": {
```

```
    "browser": true,
    "node": true,
    "mocha": true
  },
  "rules": {
    "import/no-unresolved": 0,
    "comma-dangle": 0,
    "no-console": 0,
    "no-alert": 0,
    "prefer-template": 0,
    "global-require": 0,
    "no-underscore-dangle": 0,
    "no-undef": 0,
    "react/jsx-space-before-closing": 0
  },
  "parser": "babel-eslint",
  "plugins": [
    "react"
  ]
}
```

这里引用了 Airbnb 的规范，指定了环境，禁用了一些不符合习惯的规则，并在.eslintignore 文件中指定忽略的文件夹。

.eslintignore

```
**/dist/**
**/static/**
**/build/**
**/node_modules/**
**/webpack*.js
**/webpack/**
```

最后编写 NPM 命令执行检查。

```
"lint": "eslint src"
```

不过，推荐将 ESLint 与你的编辑器集成起来，在书写代码时一直开着，这可以实时提醒你使用最佳实践来编程。主流的编辑器都包含 ESLint 的插件。

EditorConfig

为了让编辑器好好地工作，除了使用 ESLint 的插件以外，通常还需要配置 EditorConfig。EditorConfig 帮助开发者在不同的编辑器与 IDE 中使用同一种代码风格。当你格式化代码时，它会自动格式化为你定义的风格，不需要修改编辑器或 IDE 本身。启用 EditorConfig，在配置文件中定义代码风格，并下载适合自己的 EditorConfig 插件就可以了。

.editorconfig

```
root = true


[*]

# Change these settings to your own preference
indent_style = space
indent_size = 2

# We recommend you to keep these unchanged
end_of_line = lf
charset = utf-8
trim_trailing_whitespace = true
insert_final_newline = true

[*.md]
max_line_length = 0
trim_trailing_whitespace = false
```

这里根据 Airbnb 规范进行了设置。如果 EditorConfig 不能完全满足你的要求，就只能在编辑器和 IDE 中进行修改了。

总结

- 在开发环境中，通常使用开发服务器为程序提供资源服务，实现代码的热替换。开发服务器的使用避免了重复性工作，提高了开发效率。

- 前端服务器用于渲染 React 组件并吐出多个页面，使用了多种技术，包括 React 组件的服务端渲染、React Router 的服务端路由、Webpack 同构渲染、数据预载和延迟渲染、代理 API 服务器、压缩等。
- Universal 渲染的服务端指的是前端服务器，尽管可以在前端服务器中进行读/写数据库、用户认证等，但是由于前端服务器和客户端公用一套代码，所以不管出于安全的考虑，还是因为模块分离方面的因素，都应该将 API 服务器独立出去。
- 在生产环境下，应该先使用 Babel 编译 Node.js 程序，然后使用node运行。
- 在生产环境下，不需要使用开发服务器来提供资源，而是直接将其打包到静态资源目录，然后在页面中引入入口文件即可。
- 本书所有的示例程序都采用了 Airbnb 规范。它不仅规定了一些良好的代码书写习惯，还集成了一些 React 开发的最佳实践。
- 去毛机能够检查源代码中的错误以及不规范之处，提醒开发者及时修改，以此来保证代码质量。
- EditorConfig 帮助开发者在不同的编辑器与 IDE 中使用同一种代码风格。

第24章

表单

本章演示如何使用简洁的 React-Bootstrap 组件快速拼装表单，同时也会介绍如何使用 Redux-Form 做表单的验证提交工作。由于 ReudxForm 将表单状态托管在 Redux 中，所以我们能随时了解表单状态的变化。

例子

本章例子包含三个表单。

- 简单表单：单击“提交”按钮可以提交所填信息，单击“清除”按钮可以清空表单。
- 同步验证表单：在失去焦点和提交按钮时会进行字段的验证，如果字段是非法的，将在字段下面显示错误提示。
- 异步失焦验证表单：在用户名输入框失去焦点时异步验证`Username`字段，如果输入值是`john`、`paul`、`george`、`ringo`的其中一个，将会提示`That username is taken`。

每个表单在提交后，都会通过 redux-amrc 向服务器发送 POST 请求，最后在页面上显示响应结果和响应状态（见图 24-1）。

注意，从本章开始，每个示例程序都建立在前一章示例程序的基础上。

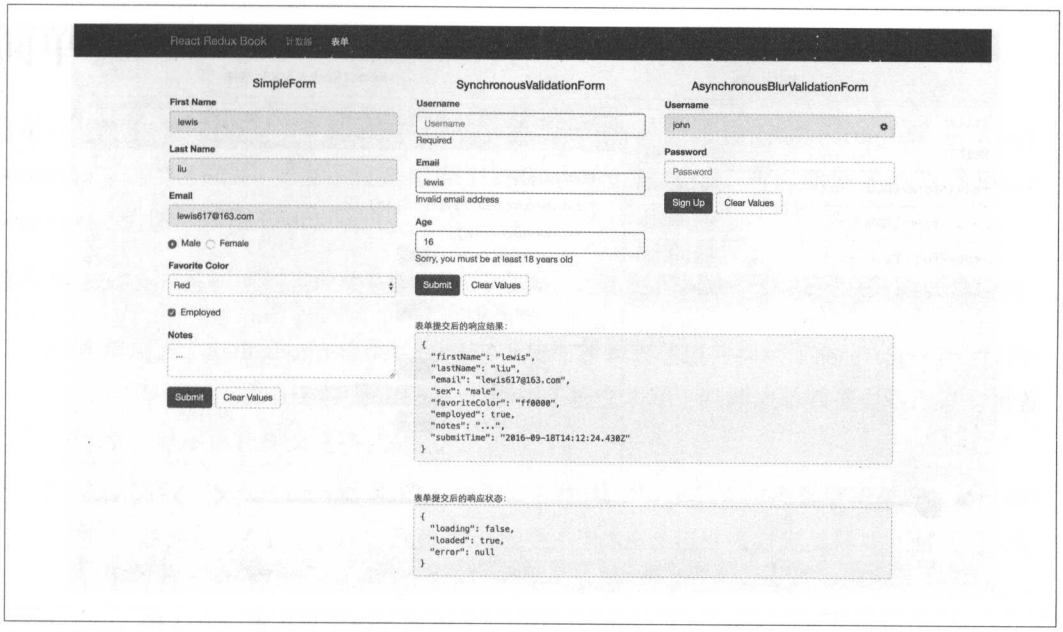


图 24-1

高阶组件 ReduxForm

ReduxForm 使用了 react-redux，让表单的状态都托管在 Redux 的 state tree 中。ReduxForm 本质上是一个高阶组件，因为它提供的装饰器reduxForm使用组件作为参数，最终生成了一个新的组件。

```
export default reduxForm({
  form: 'simple'  // a unique identifier for this form
})(SimpleForm);
```

将表单的状态放在 Redux 的 state tree 中不是必需的，因为一般情况下，不需要将这些状态共享给别的组件。但是使用 Redux 管理状态不只是为了共享方便，还为了可以清楚地了解什么时候发生什么变化。既然使用了 ReduxForm，那么就打开 Redux 开发者工具，查看 DIFF，做时间旅行吧（见图 24-2）!

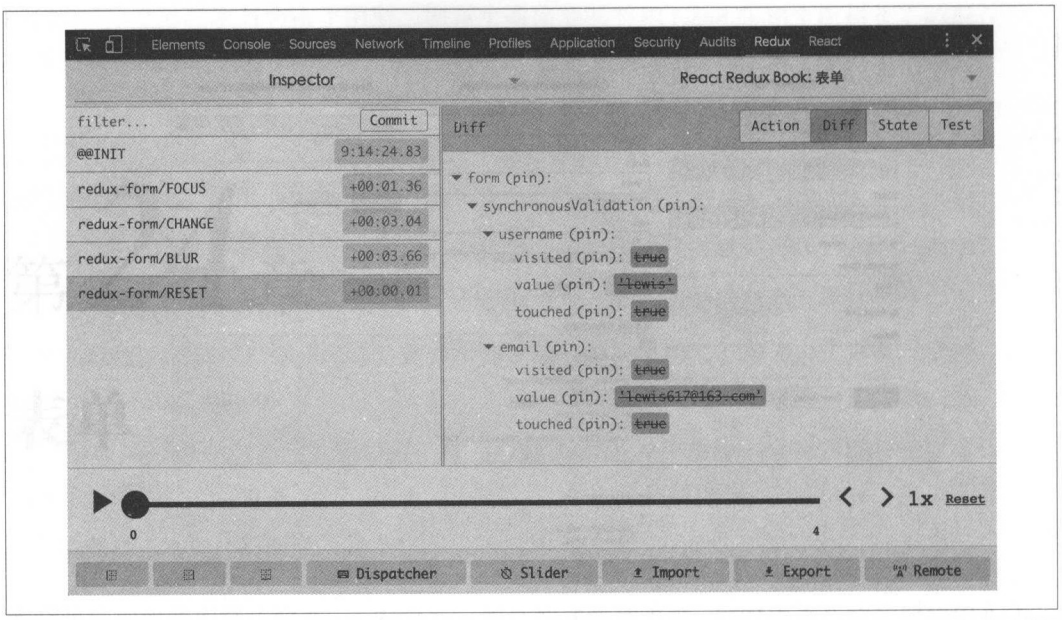


图 24-2

设置

在使用 ReduxForm 之前需要进行设置，ReduxForm 的设置非常简单，只需要添加 reducer 就可以了：

src/reducers/index.js

```
import { combineReducers } from 'redux';
import { reducer as formReducer } from 'redux-form';

const rootReducer = combineReducers({
  ...
  form: formReducer
});

export default rootReducer;
```

接下来，让我们通过“简单表单”这个例子来了解表单和 ReudxForm 是如何关联的。

简单表单

这是一个简单的例子，演示了 React-Bootstrap 的表单组件是如何与 ReudxForm 关联的。React-Bootstrap 表单组件的实质也不过是标准 HTML 表单元素而已，所以如果你不使用 React-Bootstrap，仍然可以用 ReduxForm。

那么 ReduxForm 是如何与表单关联呢？总体来说，ReduxForm 和表单的关联分为两个层面。

1. 表单层面：ReduxForm 提供的`reduxForm`装饰器为表单组件提供了`handleSubmit`、`pristine`、`reset`、`submitting`等 props，分别用于提交表单、判断表单数据是否改动、重置表单、显示正在提交这些功能。
2. 字段层面：ReduxForm 派遣了`Field`等组件作为“信使”来与表单字段关联。`Field`组件不仅提供了一个`component`属性可以接受一个 React 组件来自定义样式，而且还会给里面的表单字段传递`input`（包含`value`、`onChange`、`onBlur`、`onFocus`等）、`label`、`type`等 props 用于监听该表单字段的变化。

src/components/ReduxForms/SimpleForm.js

```
/* eslint-disable react/prop-types */
import React from 'react';
import { Field, reduxForm } from 'redux-form';
import {
  FormGroup, ControlLabel, FormControl,
  Button, ButtonToolbar, Radio, Checkbox
} from 'react-bootstrap';

const renderInput = ({ input, label, type }) => (
  <FormGroup controlId={input.name}>
    <ControlLabel>{label}</ControlLabel>
    <FormControl {...input} placeholder={label} type={type}/>
  </FormGroup>
);

const renderRadio = ({ input, label }) => (
  <Radio inline {...input} >{label}</Radio>
);
```

```
const renderCheckbox = ({ input, label }) => (
  <FormGroup>
    <Checkbox inline {...input}>{label}</Checkbox>
  </FormGroup>
);

const renderSelect = ({ input, label, children }) => (
  <FormGroup controlId={input.name}>
    <ControlLabel>{label}</ControlLabel>
    <FormControl
      componentClass="select"
      {...input}
    >
      {children}
    </FormControl>
  </FormGroup>
);

const renderTextarea = ({ input, label }) => (
  <FormGroup controlId={input.name}>
    <ControlLabel>{label}</ControlLabel>
    <FormControl
      componentClass="textarea"
      placeholder={label}
      {...input}
    />
  </FormGroup>
);


const SimpleForm = (props) => {
  const { handleSubmit, pristine, reset, submitting } = props;
  return (
    <form onSubmit={handleSubmit}>
      <Field name="firstName" component={renderInput} type="text" label="First Name"/>
      <Field name="lastName" component={renderInput} type="text" label="Last Name"/>
      <Field name="email" component={renderInput} type="email" label="Email"/>
      <FormGroup>
```

```
        <Field name="sex" component={renderRadio} type="radio" value="male" label="Male
            "/>
        <Field name="sex" component={renderRadio} type="radio" value="female" label="
            Female"/>
      </FormGroup>

      <Field name="favoriteColor" component={renderSelect} label="Favorite Color">
        <option></option>
        <option value="ff0000">Red</option>
        <option value="00ff00">Green</option>
        <option value="0000ff">Blue</option>
      </Field>

      <Field name="employed" component={renderCheckbox} type="checkbox" label="Employed
          "/>

      <Field name="notes" component={renderTextarea} type="textarea" label="Notes"/>

      <ButtonToolbar>
        <Button type="submit" bsStyle="primary" disabled={pristine || submitting}>
          {submitting ? <i /> : <i />} Submit
        </Button>
        <Button disabled={pristine || submitting} onClick={reset}>
          Clear Values
        </Button>
      </ButtonToolbar>
    </form>
  );
};

export default reduxForm({
  form: 'simple'
})(SimpleForm);
```

上述代码首先使用reduxForm装饰器给表单传递一些 props，进而控制表单的提交和清空，然后使用 React-Bootstrap 编写了一些无状态组件（renderInput等）用于表达表单中各字段的样式，并使用Field组件来关联这些字段，使得它们的value以及各种状态变化（比如onChange、onBlur、onFocus等）都能被 ReduxForm 监听到。

打开 React 开发者工具，尝试填写表单，提交表单，看看相应组件属性的变化。这个例子没有验证功能，想了解验证的实现可以学习“同步验证表单”这个例子。

同步验证表单

为了添加同步的客户端验证功能，需要在reduxForm装饰器中添加验证函数。该函数以表单数据对象values为参数，返回一个错误信息对象errors。

src/components/ReduxForms/SynchronousValidationForm.js

```
/* eslint-disable react/prop-types */
import React from 'react';
import { Field, reduxForm } from 'redux-form';
import { FormGroup, ControlLabel, FormControl, Button, ButtonToolbar } from 'react-
    bootstrap';

const validate = values => {
  const errors = {};
  if (!values.username) {
    errors.username = 'Required';
  } else if (values.username.length > 15) {
    errors.username = 'Must be 15 characters or less';
  }
  if (!values.email) {
    errors.email = 'Required';
  } else if (!/^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$/i.test(values.email)) {
    errors.email = 'Invalid email address';
  }
  if (!values.age) {
    errors.age = 'Required';
  } else if (isNaN(Number(values.age))) {
    errors.age = 'Must be a number';
  } else if (Number(values.age) < 18) {
    errors.age = 'Sorry, you must be at least 18 years old';
  }
  return errors;
};
```

```
const renderField = ({ input, label, type, meta: { touched, error } }) => (
  <FormGroup
    controlId={input.name}
    validationState={error && touched ? 'error' : null}
  >
    <ControlLabel>{label}</ControlLabel>
    <FormControl {...input} placeholder={label} type={type}/>
    {touched && error && <div className="text-danger">{error}</div>}
  </FormGroup>
);

const SyncValidationForm = (props) => {
  const { handleSubmit, pristine, reset, submitting } = props;
  return (
    <form onSubmit={handleSubmit}>
      <Field name="username" type="text" component={renderField} label="Username"/>
      <Field name="email" type="email" component={renderField} label="Email"/>
      <Field name="age" type="number" component={renderField} label="Age"/>
      <ButtonToolbar>
        <Button bsStyle="primary" type="submit" disabled={submitting}>
          {submitting ? <i /> : <i />} Submit
        </Button>
        <Button type="button" disabled={pristine || submitting} onClick={reset}>
          Clear Values
        </Button>
      </ButtonToolbar>
    </form>
  );
};

export default reduxForm({
  form: 'syncValidation',
  validate
})(SyncValidationForm);
```

这个例子的验证方法validate仅仅作为简单的演示，在实际项目中，通常会使用一个可复用的验证系统。

注意，同步验证在每次渲染时都会发生，所以如果字段数据非法，错误信息将会一直显示。如果只想在字段失去焦点和提交表单时进行验证并显示关于数据非法的提示，就要使用到touched这个标志位。

```
{touched && error && <div className="text-danger">{error}</div>}
```

打开 React 开发者工具，看看touched会在何时发生变化。

另外，React-Bootstrap 提供了validationState属性，可以根据验证状态改变组件样式，非常简洁，无须编写大量的类名。

异步失焦验证表单

有时候会要求在表单填写的过程中进行服务端异步验证，比如用户填写的用户名必须保证在系统中是唯一的，如果已存在，应该给出相应的友好提示。

为了实现异步验证，需要给reduxForm装饰器传递一个异步验证函数。该函数以表单数据对象values和dispatch为参数，返回一个 Promise 对象，该 Promise 对象在被拒绝时将返回错误信息对象，在已解决时则代表通过验证。

src/components/ReduxForms/AsynchronousBlurValidationForm.js

```
/* eslint-disable react/prop-types */
import React from 'react';
import { Field, reduxForm } from 'redux-form';
import { FormGroup, ControlLabel, FormControl, Button, ButtonToolbar } from 'react-
    bootstrap';

const validate = values => {
  const errors = {};
  if (!values.username) {
    errors.username = 'Required';
  }
  if (!values.password) {
    errors.password = 'Required';
  }
  return errors;
};
```

```
const asyncValidate = (values/* , dispatch */) =>
  new Promise((resolve, reject) => {
    setTimeout(() => {
      if (['john', 'paul', 'george', 'ringo'].includes(values.username)) {
        reject({ username: 'That username is taken' });
      } else {
        resolve();
      }
    }, 1000); // simulate server latency
  });

const renderField = ({ input, label, type, meta: { asyncValidating, touched, error } })
    => (
  <FormGroup
    controlId={input.name}
    validationState={error && touched ? 'error' : null}
  >
    <ControlLabel>{label}</ControlLabel>
    <FormControl
      type={type}
      placeholder={label}
      {...input}
    />
    <FormControl.Feedback>
      <span>
        {asyncValidating && <i className="fa fa-cog fa-spin fa-fw"/>}
      </span>
    </FormControl.Feedback>
    {touched && error &&
    (<div className="text-danger">{error}</div>)}
  </FormGroup>
);

const AsyncValidationForm = (props) => {
  const { handleSubmit, pristine, reset, submitting } = props
  return (
    <form onSubmit={handleSubmit}>
      <Field name="username" type="text" component={renderField} label="Username"/>
```

```
      <Field name="password" type="password" component={renderField} label="Password"/>
      <ButtonToolbar>
        <Button bsStyle="primary" type="submit" disabled={submitting}>
          {submitting ? <i /> : <i />} Sign Up
        </Button>
        <Button type="button" disabled={pristine || submitting} onClick={reset}>
          Clear Values
        </Button>
      </ButtonToolbar>
    </form>
  );
};

export default reduxForm({
  form: 'asyncValidation', // a unique identifier for this form
  validate,
  asyncValidate,
  asyncBlurFields: ['username']
})(AsyncValidationForm)
```

打开 React 开发者工具，看看asyncValidating标志位会在何时发生变化。另外，这里使用 React-Bootstrap 提供的FormControl.Feedback组件，来盛放异步失焦验证时的等待图标。更多 Redux-Form 的用法，可以参考官方文档（http://redux-form.com/ ）。

提交表单到服务器

为了提交表单数据到服务器，首先需要编写一个 action 创建函数：

src/actions/forms.js

```
import { ASYNC } from 'redux-amrc';
import { customFetch } from '../utils/utils';

export function post(data) {
  const key = 'forms';
  const option = {
    method: 'post',
```

```
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(data)
  };
  return {
    [ASYNC]: {
      key,
      promise: () => customFetch('/forms', option)
    }
  };
}
```

该函数使用了customFetch('/forms', option)做 POST 请求，并生成 Promise 对象，然后将 Promise 对象传给 redux-amrc，无须编写 action 和 reducer，一个完整的、基于 Redux 的异步操作就完成了，redux-amrc 会帮助我们做接下来所有的事情。

编写完 action 创建函数，制作一个盛放表单的页面，并将需要的 action 创建函数和状态绑定上去：

src/containers/Forms/Forms.js

```
import React, { Component, PropTypes } from 'react';
import Helmet from 'react-helmet';
import { Grid, Row, Col } from 'react-bootstrap';
import { connect } from 'react-redux';
import serialize from 'serialize-javascript';
import * as ActionCreators from '../../actions/forms';
import {
  SimpleForm,
  SynchronousValidationForm,
  AsynchronousBlurValidationForm
} from '../../components';

@connect(
  state => ({
    forms: state.async.forms,
    formsState: state.async.loadState && state.async.loadState.forms
  }),
```

```
  ActionCreators
)
class Forms extends Component {//eslint-disable-line
  static propTypes = {
    forms: PropTypes.any,
    formsState: PropTypes.any,
    post: PropTypes.func.isRequired
  };

  render() {
    const styles = require('./Forms.scss');
    const { forms, formsState, post } = this.props;
    return (
      <div className={styles.forms}>
        <Helmet title="表单"/>
        <Grid>
          <Row>
            <Col xs={12} md={4}>
              <h4>SimpleForm</h4>
              <SimpleForm onSubmit={post}/>
            </Col>
            <Col xs={12} md={8}>
              <Row>
                <Col xs={12} md={6}>
                  <h4>SynchronousValidationForm</h4>
                  <SynchronousValidationForm onSubmit={post}/>
                </Col>
                <Col xs={12} md={6}>
                  <h4>AsynchronousBlurValidationForm</h4>
                  <AsynchronousBlurValidationForm onSubmit={post}/>
                </Col>
              </Row>
              <br/><br/>
              <Row className="container-fluid">
                表单提交后的响应结果：
                <pre>{serialize(forms, { space: 2 })}</pre>
                <br/><br/>
                表单提交后的响应状态：
```

```
                <pre>{serialize(formsState, { space: 2 })}</pre>
              </Row>
            </Col>
          </Row>
        </Grid>
      </div>
    );
  }
}
export default Forms;
```

每个表单组件都有一个onSubmit的 props，在表单提交时被触发，参数为表单数据。这里只需要将post这个 action 创建函数传给onSubmit即可。

因为 action 创建函数将这个异步操作绑定在forms这个key上，所以应该在state.async.forms中监听表单提交的响应结果，在state.async.loadState.forms中监听表单提交的响应状态。

同时，这里还使用了 React-Bootstrap 提供的栅格组件进行栅格布局，使得组件代码非常简洁明了。

最后，编写 API 进行测试：

src/api/controllers/forms.js

```
export default app => {
  app.post('/forms', (req, res) => {
    const body = req.body;
    body.submitTime = new Date;
    setTimeout(() => {
      if (Math.random() < 0.33) {
        res.status(500).end();
      } else {
        res.json(body);
      }
    }, 1000);
  });
};
```

这里仅仅将请求体添加一个submitTime属性并返回；同时设置了有 33% 的概率会请求失败，用于测试网络请求失败的情况。

注意，为了正确解析请求体，需要引入body-parser：

```
import bodyParser from 'body-parser';
...
app.use(bodyParser.json());
```

因为这里的 POST 请求方式是application/json，所以只需要设置bodyParser.json()即可。

总结

- ReduxForm 本质上是一个高阶组件，它使用了 react-redux，让表单的状态都托管在 Redux 的 state tree 中。
- 通过reduxForm装饰器和Field组件，表单和 ReduxForm 可以关联起来，进而实现表单的提交、验证等常用功能。

第25章

图表与表格

本章将介绍如何使用 React 与 Redux 制作图表与表格。

例子

本章例子编写了两个图表组件和一个表格组件，并添加了一个统计页面，用于盛放这些组件。组件的数据均来自统计页面，而统计页面则是通过 redux-amrc 从服务器获取数据的（见图 25-1）。

图表组件

社区活跃是 React 的一个很大优点，无论你想实现什么功能，几乎都有现成的库或者组件可以使用。尽管可以自己实现图表，但是使用第三方库或许是更高效的做法。本章的示例程序将演示如何使用 React Highcharts 快速制作图表。

Highcharts 是一个功能强大、开源、美观、图表丰富，而且兼容绝大多数浏览器的纯 JavaScript 图表库。React Hightcharts 将其封装为 React 组件，只需要将 Highcharts 的参数传给组件就可以渲染出相应的图表。

首先，安装 React Highcharts：

```
npm install react-highcharts --save
```

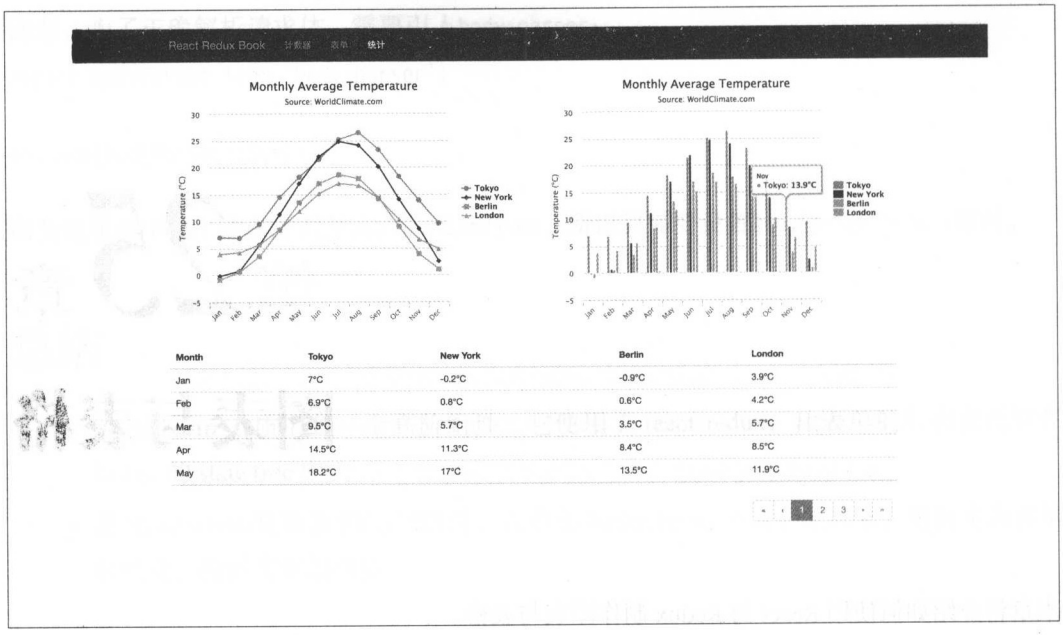


图 25-1

然后将 Highcharts 配置参数传给 React Highcharts 组件即可：

src/components/Charts/Line.js

```
import React, { PropTypes } from 'react';
import ReactHighcharts from 'react-highcharts';

function Line(props) {
  if (!props.statistic) return <p>数据异常</p>;

  const { categories, series } = props.statistic.chart;
  const config = {
    credits: {
      enabled: false
    },
    title: {
      text: 'Monthly Average Temperature',
      x: -20
    },
```

```
    subtitle: {
      text: 'Source: WorldClimate.com',
      x: -20
    },
    xAxis: {
      categories
    },
    yAxis: {
      title: {
        text: 'Temperature (°C)'
      },
      plotLines: [{
        value: 0,
        width: 1,
        color: '#808080'
      }]
    },
    tooltip: {
      valueSuffix: '°C'
    },
    legend: {
      layout: 'vertical',
      align: 'right',
      verticalAlign: 'middle',
      borderWidth: 0
    },
    series
  };
  return (
    <ReactHighcharts config={config}/>
  );
}

Line.propTypes = {
  statistic: PropTypes.any
};

export default Line;
```

上述代码做了这几件事：

- 编写一个无状态函数Line，用于渲染一个折线统计图。
- 在渲染前判断数据是否异常，如果数据异常则不渲染图表，并给出提示。
- 编写配置参数config，这里的参数和单独使用 Highcharts 时的参数一模一样，如果你之前使用过 Highcharts，那么很快就可以掌握 React Highcharts 的用法。
- 将配置参数传给ReactHighcharts组件，并返回。
- 最后，声明组件的propTypes并导出。因为 redux-amrc 在请求失败时返回的数据为空，所以这里使用PropTypes.any。

这里，简单讲解一下 Highcharts 配置参数的含义：

- credit用于设置图表版权信息。enabled用于设置是否显示版权信息。将其设置为false即可禁用版权信息。enabled默认值为true。
- title用于设置图表标题。这里填写了标题文本和水平位移。水平位移用于将标题居中显示。
- subtitle用于设置图表副标题。这里填写了副标题文本和水平位移。
- xAxis用于设置图表的 X 轴或者类别轴。这里只设置了categories，其数据来源于父级组件的传递，通过props.statistic.chart.categories获取。
- yAxis用于设置图表的 Y 轴或者值轴。这里设置了 Y 轴的标题。
- tooltip用于设置图表的数据提示框。当鼠标悬停在某点上时，以框的形式提示该点的数据，比如该点的值、数据单位等。数据提示框内提示的信息完全可以通过格式化函数动态指定。valueSuffix用于设置后缀。
- legend用于设置图标的图例说明，它是包含图表中数列标志和名称的容器。这里设置了layout、align、verticalAlign、borderWidth，使得图例说明为垂直布局，右对齐，居中，并隐藏边框。
- series用于设置图表的数据，这里数据来自于父级组件的传递，通过props.statistic.chart.series获取。

更多的 Highcharts 参数配置，可以参考官网（http://www.highcharts.com/）或 Hightcharts 中文网（http://www.hcharts.cn/）。

与折线图类似，例子中还编写了另一个无状态函数用于渲染柱状图，不再赘述。

表格组件

表格本身可以直接通过 React-Bootstrap 提供的组件实现，但是如果想在表格中实现排序、分页、过滤等功能，就要花不少精力来编写逻辑了。所幸的是社区提供了丰富的表格组件可以实现这些功能，Griddle 便是其中一个。

Griddle 是一个简单好用的表格组件。它有以下几个优点：

- 高可定制。Griddle 不会假设你的表格样式。默认情况下，Griddle 只是渲染一个带有轻量样式的 HTML 表格。如果你不喜欢默认样式，可以使用多种方式覆盖它。
- 功能丰富。Griddle 拥有排序、过滤、分页、滚动、子表显示等丰富的功能，而且未来还会添加更多的功能。
- 快速。Griddle 使用了 React 和它的虚拟 DOM 技术。默认情况下，只有当前页被渲染。数据可以一次性全部加载，也可以在服务端实现分页。所以，不管你的数据有多少，Griddle 都可以快速渲染。

安装方法：

```
npm install griddle-react --save
```

注意，Griddle 仅仅依赖于 React，和 Redux 没有关系。因此，表格的状态将无法托管在 Redux 中。在实际项目中，Redux 仅用于组件与服务器的状态交互以及多组件之间的状态共享。所以，如果你的组件状态不需要在全局共享，则不必使用 Redux。不过，如果你是 Redux 控，也可以将所有的状态全部托管在 Redux 中，让 React 组件只负责显示状态。事实上，React 结合 Redux 可以也可以快速开发一个类“Griddle”系统。

Griddle 的使用非常简单，只需要将数据传给它即可：

src/components/Table/Table.js

```
import React, { PropTypes } from 'react';
import Griddle from 'griddle-react';
import CustomPagerComponent from './CustomPagerComponent';
require('./Table.scss');

const customComponent = props => (<div>{props.data + '°C'}</div>);

customComponent.propTypes = {
```

```
  data: PropTypes.number.isRequired
};

function Table(props) {
  if (!props.statistic) return <p>数据异常</p>;
  const columnMetadata = [
    {
      columnName: 'Month'
    },
    {
      columnName: 'Tokyo',
      customComponent
    },
    {
      columnName: 'New York',
      customComponent
    },
    {
      columnName: 'Berlin',
      customComponent
    },
    {
      columnName: 'London',
      customComponent
    }
  ];
  return (
    <Griddle
      tableClassName="table table-striped"
      useGriddleStyles={false}
      results={props.statistic.table}
      columnMetadata={columnMetadata}
      useCustomPagerComponent
      customPagerComponent={CustomPagerComponent}
      resultsPerPage={5}
    />
  );
}
```

```
Table.propTypes = {
  statistic: PropTypes.any
};

export default Table;
```

上述代码做了这几件事：

- 在数据异常时，不渲染表格，并给出提示。
- 编写customComponent和columnMetadata，为所有的气温数据添加°C后缀。
- 设置 Griddle 的tableClassName，使其拥有 Bootstrap 表格的样式；还设置了 useGriddleStyles 为false，来禁用默认样式。
- 将来自父级组件的props.statistic.table传给results，作为表格数据。
- 设置 Griddle 的useCustomPagerComponent、customPagerComponent来自定义分页组件，并通过设置resultsPerPage来指定每页显示的条数。

Griddle 默认有分页功能，但是如果你想自定义分页组件，则需要传递一个新组件到 customPagerComponent，这个组件需要开发者自己编写：

src/components/Table/CustomPagerComponent.js

```
import React, { Component, PropTypes } from 'react';
import { Pagination } from 'react-bootstrap';

export default class CustomPagerComponent extends Component {
  static propTypes = {
    setPage: PropTypes.any,
    currentPage: PropTypes.any,
    previous: PropTypes.any,
    next: PropTypes.any,
    maxPage: PropTypes.any
  };
  static defaultProps = {
    maxPage: 0,
    currentPage: 0
  };
```

```
  handleSelect = (eventKey) => {
    this.props.setPage(eventKey - 1);
  };

  render() {
    return (
      <div className="pull-right">
        <Pagination
          prev
          next
          first
          last
          ellipsis
          boundaryLinks
          items={this.props.maxPage}
          maxButtons={5}
          activePage={this.props.currentPage + 1}
          onSelect={this.handleSelect}
        />
      </div>
    );
  }
}
```

分页组件的 props 是固定的，其余所有的结构样式都可以自由定制。这里使用 React-Bootstrap 提供的Pagination来替换 Griddle 默认的分页组件。注意，因为 React-Bootstrap 的Pagination与 Griddle 的分页模块在页数上差一，所以在activePage和setPage的设置上需要处理一下。

更多的 Griddle 用法，请参考官网（http://griddlegriddle.github.io/Griddle/）。

至此，表格组件就编写完成了。

从服务器获取数据

为了演示如何从服务器获取数据，首先要编写一个 API 服务，用于提供图表和表格所需要的数据。

src/api/controllers/statistic.js

```
export default app => {
  app.get('/statistic', (req, res) => {
    const categories = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
      'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
    const series = [{
      name: 'Tokyo',
      data: [7.0, 6.9, 9.5, 14.5, 18.2, 21.5, 25.2, 26.5, 23.3, 18.3, 13.9, 9.6]
    }, {
      name: 'New York',
      data: [-0.2, 0.8, 5.7, 11.3, 17.0, 22.0, 24.8, 24.1, 20.1, 14.1, 8.6, 2.5]
    }, {
      name: 'Berlin',
      data: [-0.9, 0.6, 3.5, 8.4, 13.5, 17.0, 18.6, 17.9, 14.3, 9.0, 3.9, 1.0]
    }, {
      name: 'London',
      data: [3.9, 4.2, 5.7, 8.5, 11.9, 15.2, 17.0, 16.6, 14.2, 10.3, 6.6, 4.8]
    }];

    const table = categories.map(
      (Month, key) => ({
        Month,
        Tokyo: series[0].data[key],
        'New York': series[1].data[key],
        Berlin: series[2].data[key],
        London: series[3].data[key]
      })
    );

    setTimeout(() => {
      if (Math.random() < 0.33) {
        res.status(500).end();
      } else {
        res.json({
          chart: {
            categories,
            series
```

```
        },
        table
      });
    }
  }, 1000);
 });
};
```

这个 API 所提供的categories和series和 Highcharts 官网示例程序中的一样。不过，这里还将其转换为数组table，作为表格的数据来源。与前面的 API 服务相同，这里也设置有 33%的概率会返回 500 错误。

最后，编写容器组件Statistic，用于盛放两个图表组件和一个表格组件，并使用 redux-amrc 从服务器获取数据：

src/containers/Statistic/Statistic.js

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
import { Grid, Row, Col } from 'react-bootstrap';
import Helmet from 'react-helmet';
import { Line, Column, Table } from '../../components';

@connect(
  state => ({ statistic: state.async.statistic })
)
class Statistic extends Component { // eslint-disable-line
  render() {
    const styles = require('./Statistic.scss');
    return (
      <div className={styles.statistic}>
        <Helmet title="统计"/>
        <Grid>
          <Row>
            <Col xs={12} md={6}>
              <Line {...this.props}/>
            </Col>
            <Col xs={12} md={6}>
              <Column {...this.props}/>
```

```
          </Col>
        </Row>
        <Row>
          <Col xs={12} md={12}>
            <Table {...this.props}/>
          </Col>
        </Row>
      </Grid>
    </div>
  );
 }
}

export default Statistic;
```

src/actions/statistic.js

```
import { ASYNC } from 'redux-amrc';
import { customFetch } from '../utils/utils';

export function loadStatistic() {
 return {
  [ASYNC]: {
   key: 'statistic',
   promise: () => customFetch('/statistic')
  }
 };
}
```

容器组件使用展开语法将this.props全部传给了子组件，子组件获取数据后进行渲染，整个程序就编写完成了。

总结

- Highcharts 是一个功能强大、开源、美观、图表丰富，而且兼容绝大多数浏览器的纯 JavaScript 图表库。React Hightcharts 将其封装为 React 组件，只需要将 Highcharts 的参数传给组件就可以渲染出相应的图表。

- Griddle 是一个简单好用的表格组件。它拥有高可定制性、功能丰富、快速等优点。
- 在实际项目中，Redux 仅用于组件与服务器的状态交互以及多组件之间的状态共享。

第26章
用户认证

本章将介绍如何在 React 与 Redux 的项目中实现基于 session 的用户认证。

例子

本章例子是在开源项目 react-redux-universal-hot-example 的基础上改进而成。这个项目本身非常优秀，但是更加值得推荐的是它的 Fork 项目。来自世界各地的开发者们基于该项目进行了各种改编，实现了丰富的功能，很有参考价值。虽然本章仅介绍了基于 session 的认证方法，但如果你想实现别的认证方案，比如 JSON WEB TOKEN，就可以在该项目的 Fork 项目中找到可以参考的示范。

下面介绍本章的示例程序所实现的功能。在未认证状态下，导航栏将隐藏计数器、表单和统计这三个被保护的页面的链接。如果在浏览器中输入被保护页面的地址，将会重定向到首页。如果请求被保护的 API 服务，也会返回 401，即未认证。

在登录页面输入任意用户名，单击按钮登录，将会把用户名传给 API 服务器的 session。如果当前的 session 包含用户名，则认为登录成功。

当登录成功后，将进入已认证状态。此时，导航栏会显示被保护的页面链接，进入被保护的页面也不会发生重定向。API 服务器被保护的路由也会正常返回数据。不过，已认证状态只维持一分钟。过期后，将会刷新页面，进入未认证状态。另外，单击“登出”按钮，也会从已认证状态变为未认证状态。效果图如图 26-1 和图 26-2 所示。


React Redux Book
登录
登录
输入用户名
登录
此操作将输入的用户名传给API服务器的session。

图 26-1

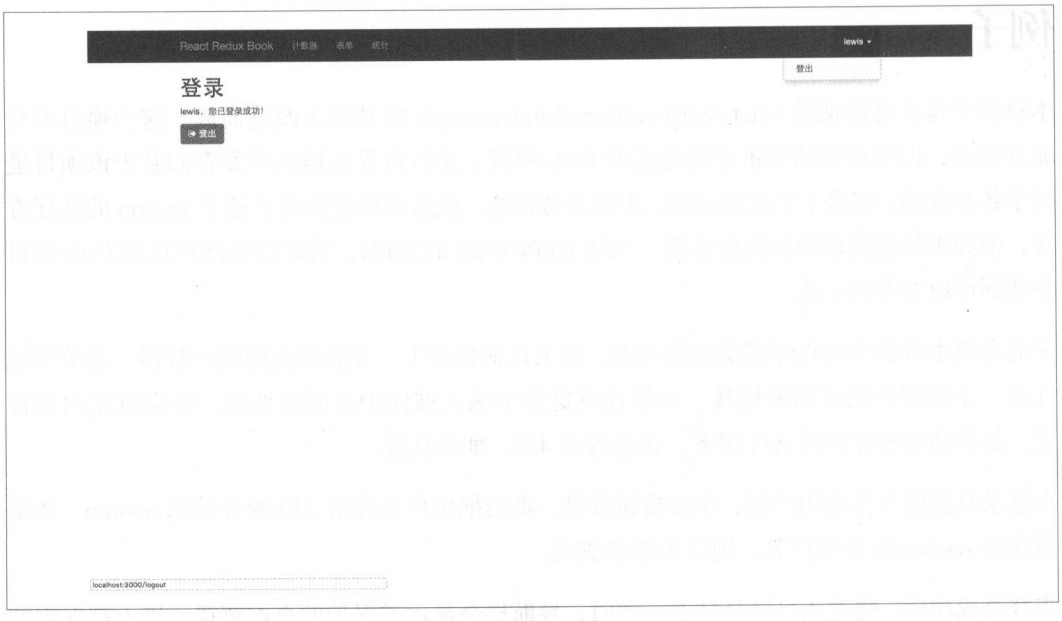

React Redux Book
计数器
表单
统计
lewis
登出
登录
lewis，您已登录成功！
登出
localhost:3000/logout

图 26-2

编写认证 API

在使用 React 与 Redux 实现客户端的认证功能前，先要编写认证 API，提供加载当前认证状态、登录、登出等操作。

Express-session 中间件

因为本章的示例程序是基于 session 进行用户认证的，所以需要安装 Express 的 session 中间件：

```
npm install express-session
```

然后指定配置选项：

src/api/api.js

```
app.use(session({
  secret: 'react redux book !!!!',
  resave: false,
  saveUninitialized: false,
  cookie: { maxAge: 60000 }
}));
```

每个参数的含义如下：

- secret是一个密钥，用于签名 session ID cookie，防止他人篡改。
- resave用于强制保存 session，即便 session 没有被修改。大多数情况下，应该将resave设置为false。
- saveUninitialized用于强制保存未初始化的 session。如果设置为false，新的 session 在被修改时才会生成；如果设置为true，新 session 将会在首次请求时生成。本例中saveUninitialized设置为false，因为只需要在进行登录请求时生成 session，发送 cookie 即可。

如果你想了解 session 的生成时刻，可以编写一个中间件，在每次请求时打印 session 和 cookie，然后尝试改变saveUninitialized的值，看看有什么区别。

加载当前认证状态

在本例中，服务器是通过判断当前 session 中是否存在user来确定登录状态的。因此，加载当前认证状态只需要返回当前 session 中的user即可。如果user不存在，则返回null。

src/api/controllers/loadAuth.js

```
export default app => {
  app.get('/loadAuth', (req, res) => {
    res.json(req.session.user || null);
  });
};
```

那么 session 中的user从何而来，在何时发生变化呢？当然是在执行登录和登出操作时。

登录与登出

登录操作是为了将认证状态从未认证变为已认证。它先从请求体中拿到用户名，存到 session 的user.name中，然后和/loadAuth一样返回user对象：

src/api/controllers/login.js

```
export default app => {
  app.post('/login', (req, res) => {
    const user = {
      name: req.body.name
    };
    const session = req.session;
    session.user = user;
    res.json(user);
  });
};
```

登出操作是为了将认证状态从已认证变为未认证，它首先销毁 session，然后返回null作为user对象，以便让客户端知道当前为未认证状态。

src/api/controllers/logout.js

```
export default app => {
  app.get('/logout', (req, res) => {
```

```
    let session = req.session;
    session.destroy(() => {
      session = null;
    });
    res.json(null);
  });
};
```

API 权限保护

在未认证状态下，只能请求未保护的 API，若请求其他 API 将会返回 401：

src/api/controllers/index.js

```
import counter from './counter';
import forms from './forms';
import statistic from './statistic';
import loadAuth from './loadAuth';
import login from './login';
import logout from './logout';

export default app => {
  loadAuth(app);
  login(app);
  logout(app);

  app.use((req, res, next) => {
    if (req.session.user) {
      next();
    } else {
      res.sendStatus(401);
    }
  });

  counter(app);
  forms(app);
  statistic(app);
};
```

编写一个处理 401 的中间件，放在被保护的 API 前面拦截请求。如果处于认证状态，则继续执行后面的 API，否则直接返回 401，表示未认证。

至此，认证 API 就编写完成了。

发送 cookie

当发出登录请求时，服务器会给 session 添加一个user对象。因为设置了saveUninitialized为false，所以此时会存储一个 session，即创建一个新 session，并产生一个 session ID。这个 session ID 将会通过请求响应存储在浏览器 cookie 的connect.sid字段中。下次再发出请求时，cookie 会携带这个字段，服务器通过 session ID 与存储中的 session 数据关联起来，就能知道当前的认证状态了。

这整个流程都没有问题，但是因为 cookie 存在浏览器中，所以前端服务器发出的请求在默认情况下是不带 cookie 的。因此需要我们手动发送 cookie。

src/utils/utils.js

```
export function customFetch(url, option) {
  const prefix = __SERVER__ ? 'http://' + config.apiHost + ':' + config.apiPort : '/api
      ';

  let opt = option || {};
  if (__SERVER__) {
    opt = {
      ...opt,
      headers: {
        ...opt.headers,
        cookie: __COOKIE__
      }
    };
  } else {
    opt = {
      ...opt,
      credentials: 'same-origin'
    };
  }
```

```
  return fetch(prefix + url, opt)
    .then(handle401)
    .then(handleErrors);
}
```

这个 customFetch 方法在原来的基础上，添加了发送 cookie 的功能。当前端服务器调用 customFetch 方法时，将会从全局变量 __COOKIE__ 中获取 cookie，放在请求头中。这个全局变量 __COOKIE__ 来自于前端服务器的中间件，每次发生请求时，前端服务器都会从浏览器中获取 cookie，传给全局变量 __COOKIE__ 。这里也可以将 cookie 存在 Redux 的全局状态中。

src/server.js

```
app.use((req, res) => {
  global.__COOKIE__ = req.get('cookie');
  ...
});
```

当浏览器调用customFetch方法时，直接设置credentials为'same-origin'，就可以向同源的 API 服务器发送 cookie 了。

编写用于认证的 action 创建函数

编写用于认证的 action 创建函数非常简单，只需要将customFetch生成的 Promise 传给 redux-amrc 即可。注意，这三个 action 创建函数的key均为user，如此一来，就可以通过监视state.async.user状态来判断当前的认证状态了。

src/actions/auth.js

```
import { ASYNC } from 'redux-amrc';
import { customFetch } from '../utils/utils';

export function shouldLoadAuth(state) {
  if (!state.async.loadState.user) return true;
  const loaded = state.async.loadState.user.loaded;
  return !loaded;
}
```

```
export function loadAuth() {
  return {
    [ASYNC]: {
      key: 'user',
      promise: () => customFetch('/loadAuth')
    }
  };
}

export function loadAuthIfNeeded() {
  return (dispatch, getState) => {
    if (shouldLoadAuth(getState())) {
      return dispatch(loadAuth());
    }
    return Promise.resolve();
  };
}

export function login(name) {
  const url = '/login';
  const option = {
    method: 'post',
    headers: {
      Accept: 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      name
    })
  };
  return {
    [ASYNC]: {
      key: 'user',
      promise: () => customFetch(url, option)
    }
  };
}
```

```
export function logout() {
  return {
    [ASYNC]: {
      key: 'user',
      promise: () => customFetch('/logout')
    }
  };
}
```

因为加载认证状态这个 action 创建函数会在每次页面跳转时执行，为了防止重复请求，这里还编写了一个`shouldLoadAuth()`和`loadAuthIfNeeded()`，用于先判断是否需要载入，然后再根据判断结果发出请求或返回一个什么也不操作的 Promise。

至此，用于认证的 action 创建函数就编写完成了。如果需要发起载入认证状态和登录/登出操作，只需要发起`loadAuthIfNeeded()`、`login()`、`logout()`这三个 action 创建函数即可。

登录页面和导航栏

编写完 action 创建函数，还需要在 React 组件中发起它们。这里先添加一个登录页面，用于发起登录和登出的 action 创建函数：

src/containers/Login/Login.js

```
import React, { Component, PropTypes } from 'react';
import { connect } from 'react-redux';
import Helmet from 'react-helmet';
import * as authActions from '../../actions/auth';

@connect(
  state => ({ user: state.async.user }),
  authActions
)
export default class Login extends Component {
  static propTypes = {
    user: PropTypes.any,
    login: PropTypes.func,
    logout: PropTypes.func
  };
```

```
handleSubmit = (event) => {
  event.preventDefault();
  const input = this.refs.username;
  this.props.login(input.value);
  input.value = '';
};

render() {
  const { user, logout } = this.props;
  const styles = require('./Login.scss');
  return (
    <div className={styles.loginPage + ' container'}>
      <Helmet title="登录"/>
      <h1>登录</h1>
      {!user &&
        <div>
          <form className="login-form form-inline" onSubmit={this.handleSubmit}>
            <div className="form-group">
              <input
                type="text" ref="username"
                placeholder="输入用户名"
                className="form-control"
              />
            </div>
            <button className="btn btn-success" onClick={this.handleSubmit}>
              <i className="fa fa-sign-in"/>{' '}登录
            </button>
          </form>
          <p>
            此操作将输入的用户名传给API服务器的session。
          </p>
        </div>
      }
      {user &&
        <div>
          <p>{user.name}, 您已登录成功！</p>
          <div>
```

```
              <button className="btn btn-danger" onClick={logout}>
                <i className="fa fa-sign-out"/>{' '}登出
              </button>
            </div>
          </div>
        }
      </div>
    );
  }
}
```

这个容器组件做了下面几件事：

- 使用 React Redux 的connect装饰器将认证状态和用于认证的 action 创建函数绑定到容器组件的 props 中。
- 编写登录表单，用于将用户输入的用户名传给 API 服务器的 session，实现登录操作。
- 编写已登录面板，在登录成功后显示当前登录的用户名，并提供按钮进行登出操作。
- 通过对user的判断，在不同的认证状态下显示不同的界面。

除了添加登录页面外，还需要更改导航栏，使其在不同认证状态下显示不同的链接，同时添加“登出”按钮，使用户可以在任意页面进行登出操作：

src/containers/Main/Main.js

```
import React, { Component, PropTypes } from 'react';
import { connect } from 'react-redux';
import { IndexLink, browserHistory } from 'react-router';
import { Navbar, Nav, NavItem, NavDropdown, MenuItem } from 'react-bootstrap';
import { LinkContainer } from 'react-router-bootstrap';
import Helmet from 'react-helmet';
import { Spin } from '../../components';
import config from '../../config';
import { logout } from '../../actions/auth';

@connect(
  state => ({ user: state.async.user }),
  { logout }
)
```

```
class Main extends Component {
  static propTypes = {
    children: PropTypes.any.isRequired,
    user: PropTypes.any,
    logout: PropTypes.func.isRequired
  };

  handleLogout = (event) => {
    event.preventDefault();
    this.props.logout().then(() => browserHistory.push('/'));
  };

  render() {
    require('./Main.scss');
    const user = this.props.user;
    return (
      <div>
        <Helmet {...config.app.head}/>
        <Navbar inverse>
          <Navbar.Header>
            <Navbar.Brand>
              <IndexLink to="/">
                <span>{config.app.title}</span>
              </IndexLink>
            </Navbar.Brand>
            <Navbar.Toggle />
          </Navbar.Header>

          <Navbar.Collapse>
            <Nav navbar>
              {user && (
                <LinkContainer to="/counter">
                  <NavItem eventKey={1}>计数器</NavItem>
                </LinkContainer>)}
              {user && (
                <LinkContainer to="/forms">
                  <NavItem eventKey={2}>表单</NavItem>
                </LinkContainer>)}
```

```
            {user && (
              <LinkContainer to="/statistic">
                <NavItem eventKey={3}>统计</NavItem>
              </LinkContainer>)}
          </Nav>
          <Nav navbar pullRight>
            {!user && (
              <LinkContainer to="/login">
                <NavItem eventKey={4}>登录</NavItem>
              </LinkContainer>
            )}
            {user && (
              <NavDropdown eventKey={5} title={user.name} id="usernameDropdown">
                <LinkContainer to="/logout">
                  <MenuItem eventKey={5.1} onClick={this.handleLogout}>登出</MenuItem>
                </LinkContainer>
              </NavDropdown>
            )}
          </Nav>
        </Navbar.Collapse>
      </Navbar>

      <Spin />

      <div>
        {/* this will render the child routes */}
        {React.cloneElement(this.props.children, this.props)}
      </div>
    </div>
  );
 }
}

export default Main;
```

前端路由保护

为了实现在未认证用户试图进入被保护页面时跳转到首页的功能。这里需要使用路由组件的onEnter属性。

src/routes.js

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';
import { Main, Home, Counter, NotFound, Forms, Statistic, Login } from './containers';
import { loadCounter } from './actions/counter';
import { loadStatistic } from './actions/statistic';
import { loadAuthIfNeeded } from './actions/auth';

const preload = promise => (nextState, replace, cb) => {
  if (__SERVER__ || nextState.location.action === 'PUSH') {
    promise().then(() => cb());
  } else {
    cb();
  }
};

export default store => {
  const counterPromise = () => store.dispatch(loadCounter());
  const statisticPromise = () => store.dispatch(loadStatistic());
  const authPromise = () => store.dispatch(loadAuthIfNeeded());
  const requireLogin = (nextState, replace, cb) => {
    const user = store.getState().async.user;
    if (!user) {
      replace('/');
    }
    cb();
  };
  return (
    <Route path="/" component={Main} onEnter={preload(authPromise)}>
      <IndexRoute component={Home}/>
      <Route onEnter={requireLogin}>
        <Route path="counter" component={Counter} onEnter={preload(counterPromise)}/>
```

```
        <Route path="forms" component={Forms}/>
        <Route path="statistic" component={Statistic} onEnter={preload(statisticPromise)
            }/>
      </Route>>
      <Route path="login" component={Login}/>
      <Route path="*" component={NotFound} status={404}/>
    </Route>
  );
};
```

上述代码在原来的基础上，添加了preload(authPromise)和requireLogin两个函数，并将其分别传给顶层路由和被保护路由的onEnter属性。

preload(authPromise)函数仅仅只是选择性地载入认证状态，“选择性”指的是如果已经载入认证状态，则无须重复载入。将其传给顶层路由的onEnter属性后，所有页面在进入前都会选择性地载入认证状态。

requireLogin函数先获取认证状态并执行判断，如果处于未认证状态将会重定向到首页，否则就正常进入该页面。

除了进行路由保护外，onEnter属性还可以进行路由的参数验证，以保证路由接收的参数都是合法的。

总结

- 基于 session 进行用户认证的 Express 程序需要安装 Express 的 session 中间件。
- 因为 cookie 保存在浏览器中，所以前端服务器发出的请求在默认情况下是不带 cookie 的。因此，需要我们手动发送 cookie。
- 利用 React Router 的onEnterAPI，不仅可以实现路由保护，还可以进行路由的参数验证和状态预载。

第27章

部署

本章讲解如何把用 React 与 Redux 搭建的程序部署到 Heroku 中。

例子

本章例子基于第 26 章的示例程序修改而成，可以直接部署到 Heroku，并添加了持续集成功能。

为了便于演示部署流程以及持续集成，本例的代码被独立到了另一个 GitHub 仓库中（https://github.com/lewis617/react-redux-heroku）。

运行方法：

```
heroku login
heroku create <your-app-name>
heroku config:set NPM_CONFIG_PRODUCTION=false
git push heroku master
heroku open
```

Heroku 平台

Heroku 是最早出现的 PaaS 提供商之一，从 2007 年起就开始运营。Heroku 平台的灵活性极高且支持多种编程语言。若想把程序部署到 Heroku 上，开发者要使用 Git 把程序推送到 Heroku 的 Git 服务器上。在服务器上，git push 命令会自动触发安装、配置和部署程序。下面各节将展开说明如何把用 React 与 Redux 搭建的 Universal 应用部署到 Heroku 上。

安装工具并登录

如果要使用 Heroku，程序必须托管在 Git 仓库中，因此本章将示例代码单独放到另一个 GitHub 仓库。另外，这里假设你已经注册了一个 Heroku 的免费账户。

如果上面的准备工作都已经完成，接下需要安装 Heroku Toolbelt 命令行工具来管理程序。安装成功后，就可以在终端中使用heroku命令了。

在 Heroku 客户端连接服务器之前，需要提供 Heroku 的账户和密码。heroku login命令可以完成这一操作：

```
heroku login
Enter your Heroku credentials.
Email: <your-email-address>
Password (typing will be hidden): <your-password>
...
```

准备程序

你可以直接通过命令行获取本章的示例程序：

```
git clone https://github.com/lewis617/react-redux-heroku.git
cd react-redux-heroku
```

和前一章的例子相比，react-redux-heroku 仅仅只是修改了一些 NPM 命令。

package.json

```
...
"start": "npm run prod",
...
```

```
"prod": "concurrently -k \"npm run start-api-prod\" \"npm run start-prod\"",
...
"heroku-postbuild": "npm run build"
```

上述代码做了以下几件事：

- 将npm start命令指向npm run prod。一般情况下，被部署到 Heroku 的程序中都会有一个 Procfile 文件，用于指定部署后的启动命令。但是，如果没有提供这个文件，会默认使用npm start进行启动。这里便选择了后者。
- 将npm run prod中的npm run build命令移到npm run heroku-postbuild中。这一步不是必需的，但是我们期望在 Heroku 进行构建后执行程序本身的构建命令，而不是在启动时才进行构建。这样做不仅可以防止启动超时导致的错误，还可以缩短每次休眠后的重启时长。

部署程序

首先需要在 Heroku 中创建一个新的应用程序，用于接收你的源代码：

```
heroku create <your-app-name>
...
```

在heroku create后面可以填写任意名称，不过要保证其他人没有使用过。如果你不指定名称，Heroku 将会为你随机生成一个名字。

上述操作完成后，一个名为heroku的远程 Git 也会被创建并关联到你的本地 Git 仓库中。你可以通过git remote和git remote get-url heroku来查看。

如果你的本地仓库中已经有一个 heroku 远程 Git，那么应该使用heroku git:remote -a <your-app-name>进行修改。

因为 Heroku 默认只安装dependencies中指定的依赖，而我们又将程序本身的构建工作放在 Heroku 中进行，所以需要连带devDependencies中的依赖一起安装。因此，必须设置环境变量NPM_CONFIG_PRODUCTION为false来使得 Heroku 同时安装dependencies和devDependencies：

```
heroku config:set NPM_CONFIG_PRODUCTION=false
```

现在就可以部署你的代码了。

```
git push heroku master
...
```

当你将代码推送到 heroku 这个远程 Git 中，Heroku 将会自动进行构建工作。当构建工作顺利完成后，就可以访问这个程序了：

```
heroku open
```

其他操作

在部署过程中，仅仅上述几个命令是远远不够的，我们也需要了解一些常用的其他操作。这些操作如下所述。

1. 查看日志

 程序生成的日志输出会被 Heroku 捕获，若想查看日志内容，可使用 `heroku logs`命令：

   ```
   heroku logs
   ```

 在测试过程中，还可以使用下述命令方便地跟踪日志文件的内容：

   ```
   heroku logs -t
   ```

 更多的用法请使用`heroku logs -h`查看。

2. 本地运行

 如果你想在本地运行程序进行测试，只需要使用`heroku local`命令即可。

 注意，在程序运行后，你会发现程序监听的端口号和使用`npm run prod`运行的程序的端口号不同。这是因为不管是本地运行还是在远程运行，Heroku 都会给程序指定端口。因此，如果你要将程序部署到 Heroku 中，就必须使用环境变量`PORT`作为端口号：

 src/config.js

   ```
   port: process.env.PORT || (process.env.NODE_ENV === 'production' ? 8080 : 3000)
   ```

3. 开启控制台

 如果你想运行一个命令或者应用程序的一个脚本，那么可以使用`heroku run`来实现。它也可以用来开启 PEPL（一个交互式的编程环境），而且这个环境还拥有程序的上下文。

   ```
   heroku run node
   ...
   > require('redux')
   { __esModule: true,
   ```

```
  createStore: [Function: createStore],
  combineReducers: [Function: combineReducers],
  bindActionCreators: [Function: bindActionCreators],
  applyMiddleware: [Function: applyMiddleware],
  compose: [Function: compose] }
>
```

除此之外，还可以使用 bash：

```
heroku run bash
~ $ ls
README.md  bin  build  node_modules  package.json  src  static  test  webpack
    webpack-assets.json
~ $ exit
exit
```

4. 定义变量

Heroku 为开发者提供了定义变量的功能。在运行时，这些变量将会被视为环境变量。前面我们定义了NPM_CONFIG_PRODUCTION为false。可以通过heroku config来查看这些变量：

```
heroku config
=== react-redux-heroku Config Vars
NPM_CONFIG_PRODUCTION: false
```

除了上述功能外，Heroku 还提供了数据库服务等丰富的功能，可以在官网了解更多的信息（https://www.heroku.com/）。

持续集成

持续集成（简称 CI）是一个极其实用的概念。基本上每次向共享服务器贡献代码时，CI 都会运行部分或全部测试。如果测试全部通过，一般什么也不会发生。反之，你就会收到一封提示邮件。当然这取决于你的配置。

目前 Node 中最流行的 CI 服务器是 Travis CI（https://travis-ci.org/）。如果你使用 GitHub，它也提供了卓越的集成支持，所以原本属于质量保证的范畴持续集成就放在本章例子中讲解。

Travis CI 的启用分为以下几个步骤：

1. 使用你的 GitHub 账户登录 Travis CI 官网。
2. 一旦登录成功，Travis CI 将会同步你的 GitHub 仓库。同步完成后，你需要在指定的仓库中激活 Travis CI。
3. 将.travis.yml 文件添加到你的仓库中，告诉 Travis CI 如何进行构建。这里的构建指的是安装依赖、运行测试的过程。

 .travis.yml

   ```
   language: node_js

   node_js:
     - "4"
     - "5"
     - "stable"

   script:
     - npm run lint
     - npm test
   ```

 这个配置文件告诉了 Travis CI 这些信息：该程序是用 Node.js 编写的，并会在4、5和stable版本中运行测试。运行测试的命令为npm run lint和npm test。
4. 添加.travis.yml 文件到 Git 中，提交并推送，以此来触发 Travis CI 的构建。
5. 检查构建页面中的状态，看看是否构建成功。另外，可以点击构建页面的图标获取链接，并显示在 GitHub 的 README.md 中，如图 27-1 所示。

 README.md

   ```
   [![Build Status](https://travis-ci.org/lewis617/react-redux-heroku.svg?branch=
       master)](https://travis-ci.org/lewis617/react-redux-heroku)
   ```

至此，一个基本的持续集成功能配置就完成了。更详细的用法，请参阅官网文档（https://docs.travis-ci.com/）。

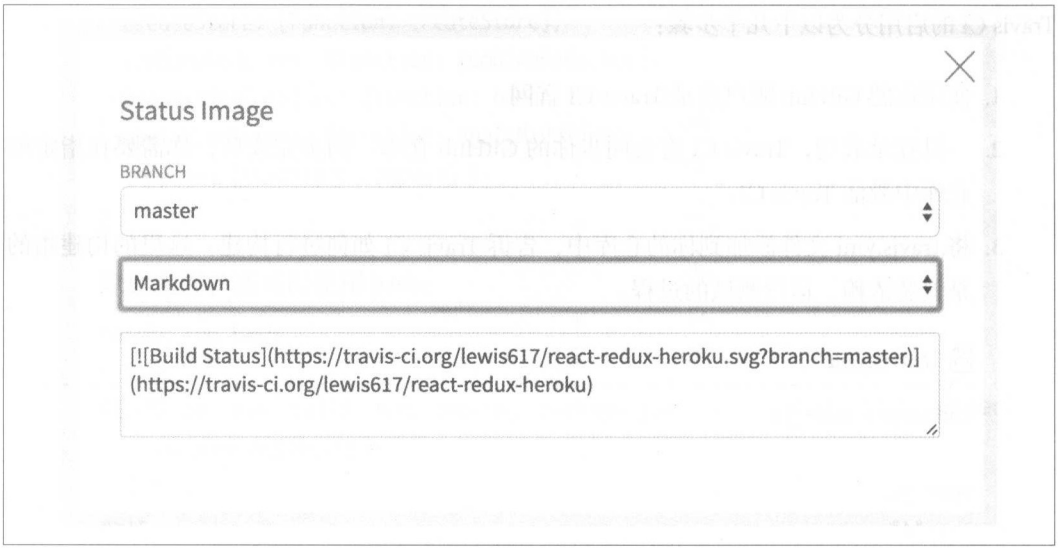


图 27-1

与生产相关的问题

由于本书的重点不在部署，因此这里讲的将程序部署到 Heroku 中，也仅仅用于实验性质的测试以及演示。在真实的项目中，往往还需要考虑更多与生产相关的问题，比如：

- 设定执行环境。在不同的环境下进行不同的配置。例如在生产环境下，不能再使用内存作为 session 的存储。
- 拓展程序。使用 Cluster 进行用应用集群扩展，以及使用代理服务器进行多台机器的拓展。
- 网站监控与压力测试。
- 安全相关。启用 HTTPS，防范跨站请求伪造攻击，采用更安全的认证系统。
- 域名注册。
- ……

总结

- 若想把程序部署到 Heroku 上，开发者要使用 Git 把程序推送到 Heroku 的 Git 服务器上。在服务器上，git push 命令会自动触发安装、配置和部署程序。
- 持续集成（简称 CI）是一个极其实用的概念。基本上你每次向共享服务器贡献代码时，CI 都会运行部分或全部测试并给出反馈。
- 将程序部署到 Heroku 中，仅仅用于实验性质的测试以及演示。在真实的项目中，往往还需要考虑更多与生产相关的问题。

第28章

其他资源

前面的章节通过文字、例子和图讲解了 React 与 Redux 的相关知识。本章作为全书的最后一章将会介绍一些有用的学习资源以及方法。授人以鱼不如授人以渔，本章虽然简短，但却可以帮助你在以后的工作与学习中，更方便地获取知识、更快速地解决问题。

在线文档

对于 JavaScript 本身而言，最好的在线文档无疑是 Mozilla 开发者网络（https:// developer.mozilla.org/）。这里面不仅包含全面详尽的 API 介绍，还给出了生动的例子。本书第 15 章的内容便是提炼自 Mozilla 开发者网络。强烈推荐大家使用这个网站。

ES2015 的相关信息可以参阅 Babel 文档中的 *Learn ES2015* 这一部分（https://babeljs.io/docs/learn-es2015/）。阮一峰的《ES 6 标准入门》也值得一读（http://es6.ruanyifeng.com/）。

作为本书阐述的最重要的几项技术，React（https://facebook.github.io/react/）、Redux（http://redux.js.org/）、Node（https://nodejs.org/api/）、NPM（https://docs.npmjs.com/）、Babel（https://babeljs.io/）与 Webpack（https://webpack.github.io/），其官网文档是必看的。除此之外，各种第三方工具的官网文档也要认真阅读。官网文档所展示的内容不仅是最权威的，而且是最新的。任何博客、书籍的更新速度都不及官网文档。尽管我会在相当长一段时间内持续更新维护本书的示例代码，但是如果你还是因为版本更新的原因遇到问题，那么官网文档则是最佳的升级参考。当然，官网文档也无法做到面面俱到。因此，与这些技术相关的博客也应该作为补充的学习资料。

参与社区活动

对于开发者而言，除了查阅在线文档外，参与社区活动，与其他同行交流也是非常有意义的。这里的“社区”可以指像 GitHub 这样的开源社区，也可以是 Stack Overflow、SegmentFault 等以问答互动为主的技术社区（当然，GitHub 也有问答功能）。

参与社区活动并不复杂，当你通过搜索引擎找到一些社区中已经存在的问答记录，并因此解决了你的问题时，就已经参与到社区活动中来了。但是，如果能够多做一些主动性强的事情或许会收获更多。这些事情包括：

- 在问答社区提问或者回答问题。
- 在某个开源项目中，提交 Issue（包括 Bug、改进建议等）或者“Pull Request”（将你做的修改放到项目中）。
- 翻译一些技术博客或者文档。
- 将自己编写的一些模块（比如中间件、工具包等）封装成 NPM 包，开源发布。
- 开源自己的程序。

如果你从来没做过上述这些事情，就尝试将本书示例代码所在的开源项目作为你参与的第一个开源项目吧。你可以将一些疑问以 Issue 的形式提交出来，也可以将一些文档说明翻译成其他语言，甚至可以编写一些 React 与 Redux 相关的示例程序作为本书的补充，我会在 Issue 或者文档中给出你的程序代码地址，以帮助更多的人。

参与社区活动并非只是做贡献，不少知名公司将应聘者在 Stack Overflow 上的声望、GitHub 上的 Star 数、技术博客的产量等社区参与经历作为判断其能力的重要参考。不仅如此，如果你能发起并运作一个成功的开源项目，每个月还可以收到非常可观的捐助支持。当然，社区最伟大的地方还是创造价值和互动互助。没有社区，就没有本书所介绍的这些优秀技术。我们从社区众多志愿者的劳动中受益，也应该考虑通过某种方式来回馈社区。

最后，真心希望 React 与 Redux 以及社区中其他优秀的技术可以造福更多的开发者。

十载耕耘奠定专业地位

博文视点诚邀精锐作者加盟

《C++Primer（中文版）（第5版）》、《淘宝技术这十年》、《代码大全》、《Windows内核情景分析》、《加密与解密》、《编程之美》、《VC++深入详解》、《SEO实战密码》、《PPT演义》……

“圣经”级图书光耀夺目，被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、咎辉Zac、李刚、曹江华……

“明星”级作者济济一堂，他们的名字熠熠生辉，与IT业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治，成就**博**古通今、**文**圆质方、**视**角独特、**点**石成金之计算机图书的风向标杆：博文视点。

“凤翱翔于千仞兮，非梧不栖”，博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟，与大师并列于IT专业出版之巅。

书为证彰显卓越品质

英雄帖

江湖风云起，代有才人出。
IT界群雄并起，逐鹿中原。
博文视点诚邀天下技术英豪加入，
指点江山，激扬文字
传播信息技术，分享IT心得

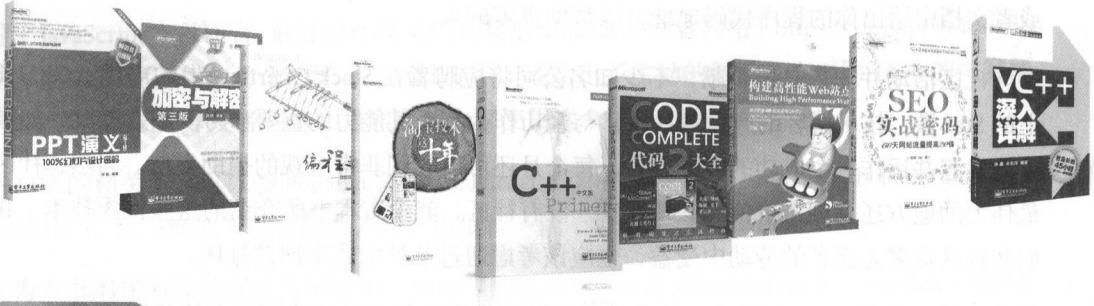
• 专业的作者服务 •

博文视点自成立以来一直专注于IT专业技术图书的出版，拥有丰富的与技术图书作者合作的经验，并参照IT技术图书的特点，打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持：

善待作者——我们会把出版流程整理得清晰简明，为作者提供优厚的稿酬服务，解除作者的顾虑，安心写作，展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯，并会参照作者实际的工作、生活节奏，量身制定写作计划，确保合作顺利进行。

提升作者——我们打造精品图书，更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力，为作者的事业开拓带来更多的机会。

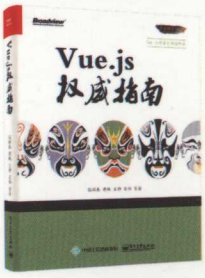


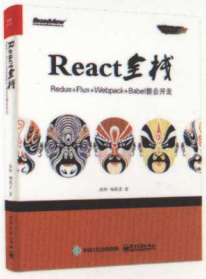
联系我们

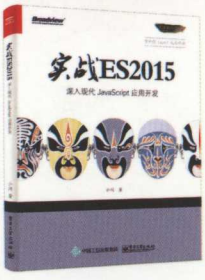
博文视点官网：http://www.broadview.com.cn　　CSDN官方博客：http://blog.csdn.net/broadview2006/

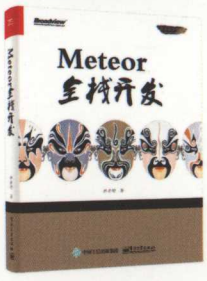
投稿电话：010-51260888　88254368　　投稿邮箱：jsj@phei.com.cn

新浪微博 weibo.com　@博文视点Broadview　　微信公众账号　博文视点Broadview


Vue.js
权威指南


React全栈
Redux+Flux+Webpack+Babel整合开发


实战ES2015
深入现代 JavaScript 应用开发


Meteor
全栈开发


React与Redux
开发实例精解

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

这本书是为想要用React、Redux、Babel 和Webpack 搭建Web 应用程序的程序员准备的。与其他技术栈相比，这个技术栈（指的是React、Redux、Babel、Webpack以及社区中众多与此相关的优秀工具）的学习成本并不低，但是其中的智慧与思想却着实令人赞叹：

◎清爽的组件代码结束了满屏HTML 标签与类名的噩梦；
◎声明式地将数据映射为界面减少了烦琐的DOM操作；
◎高阶函数、管道、柯里化等函数式编程思想的运用使得复杂逻辑的处理变得相当简单；
◎自动化构建编译技术降低了源代码与静态资源的管理成本。

随着时间的推移，技术工具或许很快会更新换代，但是这些前端工程中的最佳实践思想却不会轻易过时。如果你不仅想学习这些技术的使用方法，更想了解这些技术背后所包含的思想与智慧，那么真诚地建议你阅读本书。

本书使用的示例代码可在GitHub上获取：
https://github.com/lewis617/react-redux-book和
https://github.com/lewis617/react-redux-heroku。

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

责任编辑：许　艳
封面设计：吴海燕

上架建议：Web开发/前端
ISBN 978-7-121-30267-1
9 787121 302671 >
定价：59.00元